A.E. Stanley
Il BASIC 7.0
per il Commodore 128
McGraw-Hill

Il BASIC 7.0
per il Commodore 128

A.E. Stanley

Il BASIC 7.0 per il Commodore 128

McGRAW-HILL Libri Italia srl

Milano · New York · St Louis · San Francisco · Amburgo · Auckland
Bogotá · Città del Guatemala · Città del Messico · Johannesburg
Lisbona · Londra · Madrid · Montreal · Nuova Delhi · Panama · Parigi
San Juan · San Paolo · Singapore · Sydney · Tokyo · Toronto

Indice

Nota sui programmi

I programmi riportati nel testo sono in genere riprodotti direttamente per evitare errori di ricopiatura. Per migliorarne la leggibilità, però, i caratteri di controllo formato (colore del carattere, frecce, ecc.) sono spesso stati omessi: ciò significa che se vengono ricopiati dal libro il risultato può non essere elegante.
Ad eccezione dei programmi grafici, tutti i programmi possono essere usati sia con lo schermo a 40 colonne sia con quello a 80 (RGBI). Nei casi in cui è necessario o preferibile uno particolare dei due formati, se il calcolatore è nell'altro modo operativo, viene visualizzato un messaggio che richiede il cambiamento. Ciò si ottiene premendo l'apposito tasto sul monitor.
È disponibile un dischetto (vedere Appendice A) che contiene la quasi totalità dei programmi illustrati e discussi nel testo.
Il disco è un disco BOOT. L'avviamento avviene quindi in uno dei due modi seguenti: accendere il disk drive e inserire il disco, quindi accendere il calcolatore; in alternativa, a calcolatore già acceso, inserire il disco nel drive, poi battere sulla tastiera la parola BOOT seguita da RETURN. (Con il C-128D, inserire il dischetto a sistema spento, poi accendere).
Verrà visualizzato un menu dei programmi. Ogni programma, quando termina, consente all'utente di ritornare al sotto-menu per il tipo di programma richiesto.
Nel testo si sono seguite le seguenti convenzioni:

- nei programmi (cioè quando la riga inizia con un numero), il testo in BASIC è riportato in maiuscolo e in minuscolo a seconda delle esigenze del programma

• quando invece si tratta di comandi da inserire in modo diretto da tastiera, i comandi sono generalmente in maiuscolo.

In ogni caso sarà sufficiente che il lettore tenga presente che le parole chiave del BASIC non devono *mai* essere battute tenendo premuto il tasto SHIFT o con i tasti CAPS LOCK o SHIFT LOCK abbassati, se non quando si usano le forme abbreviate, come per esempio bO invece di BOOT.
Altre indicazioni relative alle convenzioni seguite si trovano all'inizio del Dizionario del BASIC 7.0.

Introduzione

La grande promessa

Forse, tra un secolo, gli anni Ottanta saranno ricordati come il periodo in cui l'informatica divenne "personale". Dalla sua nascita, immediatamente dopo la seconda guerra mondiale, fino alla fine degli anni Settanta, l'uso del calcolatore si era esteso a tutto il mondo del lavoro, dalla produzione industriale alla contabilità, alla ricerca scientifica e alla navigazione: ma erano macchine troppo complesse da usare e comunque troppo costose perché una piccola azienda o una singola persona potesse concedersi il lusso di acquistarne una.
Se questo decennio sarà ricordato per la nascita dell'informatica "personale", è difficile stabilire se il simbolo di questa rivoluzione sarà il Personal Computer IBM o il Commodore 64. Il PC offriva per la prima volta un sistema operativo che consentiva a una persona inesperta di usare un calcolatore senza molto addestramento, e una memoria sufficientemente grande per un uso rapido ed efficiente. Questo sistema operativo, MS-DOS o PC-DOS, nacque esattamente all'inizio del decennio, e la sua nascita diede in un certo senso i natali alla prima generazione di una nuova famiglia di calcolatori personali, più piccoli del PC stesso.
Gli home computer devono la loro origine al PC che utilizzò microprocessori di nuova "generazione": i microprocessori della generazione precedente, da quel momento, cominciarono a costare molto meno e furono largamente impiegati sugli home.
In particolare i microprocessori della serie 6500 diedero vita a calcolatori come il VIC-20 e il Commodore 64 e, non molto più tardi, il glorioso Zi-

log Z80 ha trovato una seconda giovinezza in una serie di calcolatori che utilizzano il sistema operativo CP/M.
Il Commodore 128, che nel seguito verrà chiamato più brevemente C-128, è un sistema basato sia sulla serie 6500 (come il C-64), sia sullo Z80. Malgrado il suo basso costo, è un calcolatore complesso e versatile che riunisce tutte le possibilità già collaudate del C-64 e del CP/M a quelle, in buona parte ancora inesplorate, del C-128 vero e proprio.
Questo libro non tenta nemmeno di esplorare tutte queste possibilità: a parte i pochi paragrafi che seguono, si dedica esclusivamente al BASIC 7.0 del C-128 e al funzionamento del calcolatore in modo 128.

Il Commodore 64

Nei pochi anni della sua esistenza è nata per il C-64 una quantità di programmi tale che sarebbe totalmente impossibile compilarne un catalogo. Questi programmi sono di tutti i tipi, anche se in gran parte sono videogiochi o altri programmi di svago; ma esistono anche molti (e ottimi) fogli elettronici, word processor, database e programmi grafici, nonché programmi più specializzati che vanno dal calcolo dell'equilibrio statico degli edifici, alla contabilità aziendale, al sistema di controllo del riscaldamento domestico.
Chi possiede un C-128 dispone anche di un C-64. È sufficiente accendere il C-128 tenendo premuto il tasto Commodore (il tasto più in basso a sinistra), oppure, con il calcolatore già acceso, inserire il comando GO 64. Quest'ultimo comando è necessario se si utilizza un disk drive modello 1571. In entrambi i casi si ha da quel momento un C-64, con tutte le sue possibilità e con tutti i suoi limiti.
Le possibilità sono tutte connesse con la vasta disponibilità di software (programmi) a prezzi ragionevoli e collaudato da una vasta utenza.
Le limitazioni, invece, riguardano la possibilità, per l'utente non professionale, di scrivere i propri programmi.
Quando il C-64 apparve sul mercato, fu acclamato perché offriva all'uomo della strada la possibilità di creare i propri programmi. Ma il BASIC del C-64 (BASIC 2.0) è in realtà molto limitato. Per un certo periodo di tempo, anche in Italia, è stato di moda iscrivere i propri figli (o se stessi) a un corso di BASIC, e nella stragrande maggioranza dei casi il calcolatore usato era il C-64. Quella moda è finita, forse perché gli studenti hanno scoperto che, purtroppo, gli strabilianti effetti grafici e sonori offerti dai videogiochi non possono essere prodotti usando il BASIC del C-64. E forse anche perché il BASIC 2.0 è troppo lontano dai BASIC dei PC (BASIC A o GW-BASIC, per esempio) per costituire una buona preparazione al loro uso.

Un'osservazione fondamentale relativa al BASIC è la seguente: è un linguaggio che, anche sacrificando altre qualità come la velocità e l'economia di memoria, è composto di comandi "umanoidi", cioè parole chiave abbastanza simili al linguaggio umano perché si possano capire e ricordare facilmente.
Per fare un esempio: usando il BASIC 2.0 per colorare il bordo dello schermo di nero è necessario usare l'istruzione

```
10 poke 53280,0
```

Ciò non è certamente difficile, ma i due numeri non sono facili da ricordare. Mentre scriviamo un programma, il nostro lavoro può essere rallentato dalla necessità di consultare una tabella per trovare i due numeri. Questo in genere non è un problema: ma se, dopo alcuni mesi, rileggiamo il nostro programma, dovremo tornare alla tabella per capire di che cosa si tratti. Il BASIC 7.0 del C-128 offre invece la possibilità di scrivere

```
10 color 4,1
```

Ma questo esempio, anche se risulta evidente la maggior comprensibilità, non è ancora nulla. Supponiamo di voler disegnare un cerchio rosso al centro dello schermo grafico ad alta risoluzione. Il BASIC 7.0 lo fa così:

```
10 graphic 1,1: color 1,3: circle 1,160,100,100: paint 1,160,100,1
```

Questo significa "SCHERMO GRAFICO AD ALTA RISOLUZIONE, PULITO - USA UNA LINEA ROSSA - DISEGNA UN CERCHIO AL CENTRO CON UN RAGGIO DI 100 PIXEL - RIEMPILO DI ROSSO".
Le istruzioni del BASIC 2.0 necessarie per ottenere lo stesso effetto occuperebbero una pagina intera: perciò rinunciamo a riprodurle in questa sede. Il fatto da tener presente è che nemmeno una sola istruzione sarebbe riconoscibile come comando grafico: si avrebbe soltanto una serie di PEEK e POKE seguiti da numeri.
PEEK e POKE sono naturalmente comandi del linguaggio BASIC. Ma tenendo presente che il BASIC è stato inventato come linguaggio utilizzabile da non professionisti, esso non dovrebbe fare largo uso di questi comandi, che in effetti dovrebbero servire per risolvere problemi rari ed insoliti. Consentono infatti di programmare in linguaggio macchina, ma nel modo più lento ed inefficiente.
Non si vuole certo negare l'importanza che il BASIC 2.0 ha avuto nella diffusione dell'informatica; ma si vedrà in questo libro che praticamente tutto ciò che sul C-64 era "programmazione avanzata" è "programmazione elementare" per il BASIC 7.0.
Il C-64 è infatti una macchina largamente superata, che rispetto al C-128

presenta soltanto il vantaggio del prezzo inferiore. Questa osservazione è valida anche se è stata immessa sul mercato una versione nuova, il C-64C, che ha soltanto una linea più elegante rispetto ai modelli precedenti: il BASIC è sempre la versione 2.0.
Nei primi quindici mesi della sua vita, il C-128 è stato venduto in circa 800.000 esemplari in tutto il mondo. Ne esistono due modelli principali: il C-128 "normale" e il C-128D, un modello portatile con un disk drive 1571 incorporato.
Come si è già sottolineato, il C-128 è "compatibile verso il basso": è capace di utilizzare tutti i programmi che sono stati scritti per il C-64. Naturalmente, però, tutti questi programmi sono scritti in modo da occupare una memoria di meno di 64 K (kilobyte o migliaia di byte). Molti videogiochi non verranno mai riscritti per il C-128, dal momento che ogni C-128 li può utilizzare grazie al comando GO 64.
I programmi applicativi più "seri" (fogli elettronici, word processor, ecc.) invece, che richiedono più memoria, diventano molto più utili in modo 128, e in effetti il software già disponibile è costituito largamente da applicazioni di questo genere, spesso di altissima qualità.
Chi possiede il C-128 può utilizzare in modo 64 una vasta gamma di software. Oltre a questo, e alla crescente produzione di software serio e valido in modo 128, esiste una terza fonte — quasi inesauribile — di programmi, grazie al sistema operativo CP/M.

Il CP/M

Quando si inserisce nel drive il disco CP/M (che viene fornito insieme al C-128) e si accende il calcolatore, quest'ultimo diventa un calcolatore completamente diverso. Il controllo del calcolatore viene affidato al microprocessore Z80 anziché al 6502: vari tasti assumono nuove funzioni; il disk drive funziona in modo diverso, e così via. Per chi è abituato al C-64, il fatto più sconcertante è che non esiste più il linguaggio BASIC, sostituito da un numero limitato di comandi del sistema operativo. Non essendovi un linguaggio residente, chi vuole scrivere un programma deve acquistare un interprete o un compilatore. Il linguaggio prescelto potrebbe ancora essere il BASIC (ne esistono diverse versioni), ma sono disponibili moltissimi altri linguaggi, dall'ANSI FORTRAN al Pascal, al linguaggio C.
Il CP/M può quindi servire per imparare un linguaggio da usare a livello professionale, poiché in genere le versioni dei vari linguaggi disponibili in CP/M sono praticamente identiche a quelle disponibili per il PC. Ma se il lettore ha propositi di questo genere vale la pena di osservare che sono disponibili vari linguaggi anche per il C-128 in modo 128. In particolare

esistono due eccellenti compilatori per il linguaggio C e almeno uno per il Pascal.
Sembrerebbe opportuno preferire questi, anche perché in genere i linguaggi disponibili in CP/M non sono dotati di comandi grafici. Ma l'uso più valido del CP/M, come del modo 64, è relativo allo sfruttamento della vasta disponibilità di software esistente sul mercato. Buona parte del software classico del PC esisteva già per calcolatori a 8 bit con il CP/M.
Chi dispone del drive 1571 a doppia faccia può trovare, anche in Italia, un buon numero di programmi. Il 1571 è in grado di leggere sia il formato GCR (Group Code Recording) della Commodore, sia vari altri formati tipici del CP/M (anche se è in grado di formattare un disco solo in formato GCR).
Chi invece dispone del vecchio drive 1541 a singola faccia riceverà poche soddisfazioni dal CP/M, sia per la quasi totale mancanza di software in questo formato, sia per la sua esasperante lentezza. Il CP/M, infatti, fu sviluppato quando un calcolatore con una memoria RAM di 64 K era considerato molto potente, e di conseguenza utilizza il dischetto molto intensivamente: anche l'operazione più semplice richiede in genere almeno un'operazione di lettura o di scrittura sul disco. Ed è difficile immaginare un drive più lento e meno efficiente del 1541.

Il modo 128

Qualcuno potrebbe a questo punto sospettare che il C-128 offra questi due modi alternativi di funzionamento per sopperire alle deficienze del modo 128 vero e proprio.
Grazie al C-64 incorporato e al CP/M, il C-128 è forse unico tra tutti i calcolatori mai prodotti per quanto riguarda la quantità di software disponibile. Ma la versatilità e la facilità d'uso del calcolatore in modo 128 lo rendono preferibile, sia per programmare nei vari linguaggi disponibili, sia per vari usi professionali. Per esempio questo libro è stato scritto con un C-128D, grazie a uno dei vari word processor di ottima qualità offerti dal mercato.
Oltre al BASIC 7.0, argomento fondamentale di questo libro, il C-128 offre due possibilità importanti: una velocità doppia rispetto a quella del C-64 e lo schermo a 80 colonne.
Il comando FAST porta la velocità di elaborazione (clock rate) da 1 MHz (megahertz) a 2 MHz. Questa non è una velocità eccezionale: un PC normale opera ad oltre 4 MHz, e velocità di 7-8 MHz sono oggi frequenti. Eppure il microprocessore della serie 6500 usato dal C-128, anche se non è ormai l'ultimo ritrovato della tecnica moderna, ha un'architettura molto semplice ed efficiente, tale che, per uno stesso comando, questa CPU a 8 bit deve esegui-

re in media un numero di operazioni inferiore a quello richiesto da un microprocessore a 16 bit come l'8088 del PC. In altre parole, se avesse lo stesso clock rate del PC, sarebbe più veloce di quest'ultimo.
Inoltre, la questione della velocità è trascurabile in molte applicazioni. Per esempio, se questo libro fosse stato scritto usando un calcolatore un milione di volte più veloce del C-128, non sarebbe arrivato in libreria nemmeno un giorno prima. Il fattore determinante è stata la velocità del cervello dell'autore, e non certo quella del C-128.
Usando un comune televisore è possibile visualizzare sullo schermo una riga di soli 40 caratteri: se si cerca di usare caratteri, più piccoli questi diventano illeggibili. Tutti i calcolatori moderni utilizzano monitor o terminali capaci di visualizzare almeno 80 caratteri per riga, possibilità quasi indispensabile per l'uso di un foglio elettronico o di un word processor.
Oltre all'uscita video in radiofrequenza, collegabile all'antenna di un TV, e all'uscita "video composito" (collegabile a un monitor o alla presa SCART di un TV moderno), il C-128 presenta un'uscita RGBI (Red Green Blue Intensity), che può essere collegato a un monitor RGBI digitale. Si ottiene così un'immagine molto più precisa, che consente appunto la visualizzazione di 80 colonne di testo. Naturalmente questo richiede un monitor RGBI, anche se esistono cavi speciali che consentono di ottenere le 80 colonne (senza il colore) anche su un monitor non RGBI. Il monitor RGBI della Commodore è ottimo, ma sono disponibili monitor analoghi molto meno costosi. Quasi tutti consentono di visualizzare anche lo schermo "tradizionale" a 40 colonne, e questo è necessario dal momento che il C-128 richiede questo schermo per la grafica.
È infine da notare che il comando FAST, che raddoppia la velocità, rende impossibile l'uso dello schermo a 40 colonne. Questo infatti si svuota, ed è necessario un comando SLOW perché si torni a vedere qualcosa. Questo non è inaccettabile: per esempio il programma ALFASORT sul Lato 1 del disco che accompagna questo volume ne fa uso per una serie piuttosto lunga di elaborazioni; significa attendere davanti a uno schermo vuoto ma il tempo d'attesa è ridotto della metà. È però ovvio il vantaggio dello schermo a 80 colonne, che diventa indispensabile quando un programma deve comunicare con l'utente mentre opera in modo veloce.

Il C-128 e l'unità dischi

Il DOS (Disk Operating System) impiegato dalla Commodore non è certo meraviglioso, né come velocità né come efficienza. Chi ha usato il vecchio drive 1541 con il C-64 ne conosce la lentezza e la complessità dei comandi. Insieme al C-128 è nato il drive 1571 che è da 5 a 12 volte più veloce e

può accedere ad entrambe le facce del disco. È nettamente superiore al drive 1541, tanto che la differenza di prezzo non deve essere un fattore determinante nella scelta.
Sinceramente bisogna dire che anche il 1571 non è meraviglioso. Il suo DOS, come quello del 1541, contiene numerosi errori, tra i quali il più grave (si veda il Dizionario del BASIC), riguarda l'uso di DSAVE e SAVE con il simbolo @ per sostituire un file. Come si è già osservato, può leggere vari formati di dischi CP/M oltre al formato Commodore, ma non è in grado di leggere i dischi MS-DOS che sono quasi uno standard mondiale. Anche se negli Stati Uniti il mercato offre almeno un programma che consente di leggere dischi MS-DOS, i tempi sono sicuramente maturi per un disk drive Commodore con un DOS moderno che consenta il trasferimento di dati ad un PC qualsiasi senza problemi.
Sul C-128 appaiono le parole Personal Computer. Viene naturale domandarsi se sono giustificate. Tralasciando il fatto che oggi si tende ad associare questo nome ai PC del tipo IBM, e quindi al sistema operativo MS-DOS, potremmo definire PC qualsiasi calcolatore capace di rispondere alle esigenze di un individuo che ha bisogno, anche per motivi professionali, di trattare più o meno quotidianamente una quantità di dati non enorme, ma tale che l'uso della macchina consenta un rilevante risparmio di tempo e/o di denaro.

- Dal punto di vista dell'hardware possiamo rispondere facilmente: il C-128 ha molta più memoria dei primi modelli del PC IBM, e la sua efficienza, in termini di velocità e di affidabilità, è ottima.
- Disponibilità di hardware ausiliario: esistono ottime stampanti di ogni tipo e di numerose marche, nonché plotter, mouse, penne ottiche, tavolette grafiche, modem.
- Affidabilità: i guasti sono infrequenti, ma il tempo richiesto per una riparazione è, in Italia almeno, assolutamente imprevedibile. Non risulta possibile ottenere un altro C-128 per il periodo della riparazione stessa.
- Disponibilità di software: come si è già detto, è considerevole; ma almeno in Italia è difficile trovare un rivenditore che ne offra un buon assortimento e ancora più difficile trovarne uno che conosca i prodotti che vende.
- Documentazione e informazioni: la documentazione fornita insieme al calcolatore non è pessima, ma non è completa. Questo libro è stato scritto facendo largo uso del manuale in lingua inglese (System Guide); le altre fonti disponibili hanno consentito di integrare le informazioni con altre più dettagliate, e molte altre notizie sono frutto di esperimenti personali da parte dell'autore.
- Assistenza: le garanzie vengono rispettate e la qualità del servizio è buona, tranne per quanto riguarda il tempo richiesto. Per quanto riguarda le informazioni, invece, l'acquirente si trova abbandonato a se

stesso. La Commodore Italiana non ha saputo fornire alcuna informazione utile alla compilazione di questo libro; la nostra impressione è che non rientri nella sua politica di vendita conoscere i prodotti che distribuisce in Italia, ma che preferisca vendere "a scatola chiusa". Oltre alla Commodore stessa, esiste ovviamente una rete di vendita costituita in buona parte da negozi non specializzati il cui personale conosce allo stesso livello di approfondimento aspirapolvere, frullini e calcolatori.

Il C-128 non è certamente un giocattolo, ma viene venduto come se lo fosse. Esistono anche negozi specializzati che dispongono di persone abbastanza esperte. Sarebbe opportuno che il pubblico preferisse questi e che fosse piuttosto esigente, richiedendo informazioni precise e un assortimento di software sempre più completo.
Per riassumere: il minor costo del C-128, rispetto al PC classico, all'Apple Macintosh o all'Amiga (per quest'ultimo valgono però tutte le osservazioni fatte sopra) lo renderebbe ideale per numerose applicazioni serie a livello personale o di piccola azienda, se in qualche modo si potesse contare su una rete di informazioni e di servizi. In mancanza di questa, il C-128 è un buon acquisto:

- per chi ha trovato, nella propria città, un negozio serio
- per chi non ha problemi di perdere tempo nella ricerca di informazioni e di software e può attendere quando è necessaria una riparazione.

Queste sono osservazioni dure ma necessarie. Se la lettura di questo volume può servire a dimostrare l'utilità del C-128 per serie applicazioni di lavoro, toccherà poi al lettore esigere, da chi lo vende, un servizio serio.

Il libro e il disco

Questo libro è nato con lo scopo preciso di offrire una "seconda possibilità" a chi aveva abbandonato lo studio del BASIC 2.0 del VIC-20 o del C-64 a causa dei limiti di questo linguaggio.
Nell'offrire questa possibilità, fornisce anche una visione d'insieme del calcolatore, senza cercare di duplicare il contenuto della System Guide. Per offrire una trattazione seria dell'argomento principale, tralascia completamente altri argomenti affascinanti, come il CP/M, la programmazione in linguaggio macchina e il sistema operativo GEOS. Quest'ultimo non era disponibile per il C-128 al momento in cui si terminava questo libro: offre un modo di usare il calcolatore — con mouse ed icone — del tutto simile al famoso Apple Macintosh. È comunque dotato di una buona do-

cumentazione propria. Sistemi di questo genere, comunque, sembrano più interessanti per chi non ama l'uso della tastiera (sostituita largamente dal mouse) e si interessa più all'uso del calcolatore — per disegnare, per scrivere testi, e così via — che non alla programmazione.
Il libro si divide in due parti principali: una serie di capitoli dedicati alle varie possibilità nuove e più importanti offerte dal BASIC 7.0, e il Dizionario del BASIC.
Mentre i capitoli centrali presuppongono una certa conoscenza del BASIC 2.0 del VIC-20 o del C-64 (e alcune parti saranno fin troppo facili per chi conosce il BASIC del C-16 o del Plus/4), il Dizionario tratta tutte le voci del BASIC 7.0 in modo che chi non conosce il BASIC 2.0 possa capire tutto il libro insieme al disco.
Chi è appena passato dal C-64 al C-128 potrà quindi leggere libro e disco come vuole: sia leggendolo dall'inizio alla fine, oppure qua e là, troverà tutte le spiegazioni necessarie.
Per chi non ha una solida conoscenza del BASIC sarà necessario usare più sistematicamente il Dizionario, anche se è meglio non leggerlo dall'inizio alla fine in ordine alfabetico perché una memoria umana difficilmente potrebbe accogliere tutto in questo modo. Un buon sistema può essere seguire i programmi (soprattutto i tutorial del Lato 1 del disco) con il libro aperto alla sezione Dizionario. Chi invece ha appena divorziato da un calcolatore dotato di un BASIC non Commodore (MSX o Sinclair, per esempio), potrebbe trovare utile il consiglio di leggere rapidamente il Dizionario per vedere quali siano le principali differenze. A titolo d'esempio i seguenti comandi MSX riveleranno differenze significative:

BSAVE, BLOAD, CIRCLE, CLOSE, COLOR, DRAW, GET, IF, INPUT, KEY, LOAD, OPEN, PAINT, RETURN, RUN, SAVE, SOUND, STOP, SWAP, WAIT, WIDTH.

Chi si è separato, probabilmente senza troppi rimpianti, dal C-16 o dal Plus/4, avrà meno da imparare. Molti comandi del BASIC 7.0 gli saranno già noti. Esistono alcune differenze importanti, come per esempio la necessità di specificare la SCALE dopo ogni comando GRAPHIC se non si desidera tornare al default SCALE 0, o la struttura completamente diversa di SOUND.
Il BASIC 7.0, per la sua completezza e per la sua flessibilità, costituisce senz'altro una buona preparazione alle versioni professionali del BASIC, come MBASIC, GW-BASIC e BASIC A. Chi, in un futuro più o meno lontano, dovesse trovarsi nella necessità di apprendere uno di questi linguaggi, pur dovendo imparare diversi comandi nuovi e rivedere la sintassi di altri, si troverà in grado di scrivere programmi semplici fin dal primo istante e sarà in grado di impadronirsi completamente del nuovo linguaggio in pochissimo tempo.

Il disco

A questo volume si accompagna un floppy disk contenente numerosi programmi, tutti studiati per facilitare l'apprendimento del BASIC 7.0. Sul Lato 1 quasi tutti i programmi sono "tutorial", cioè lezioni vere e proprie, molte delle quali comprendono anche routine che possono essere copiate ed adattate ad altri programmi. Il Lato 2 è dedicato interamente all'esercitazione nell'uso dei comandi grafici.
Al lettore che non dispone di un disk drive potrà dispiacere che si tratti di un disco e non di una cassetta. Ma il C-128, come qualsiasi calcolatore moderno, richiede per ogni uso serio il floppy disk. Varie operazioni previste dal BASIC 7.0 sono possibili soltanto con il dischetto, e per illustrare queste era impossibile usare la cassetta. Perciò si può soltanto sottolineare che un C-128 sprovvisto di floppy disk è come un'automobile senza pneumatici, e consigliare al lettore di custodire il disco in luogo sicuro fino al giorno in cui acquisterà il drive.
Un'appendice alla fine di questo volume fornisce informazioni specifiche sul contenuto dei due lati del disco, nonché le istruzioni necessarie per preparare le "copie di lavoro" dei programmi grafici sul Lato 2. Questo lato, infatti, comprende programmi che devono essere in grado di registrare disegni e dati sul disco stesso. Ciò significa che ogni disco deve disporre di uno spazio libero. È quindi necessario creare vari dischi più specializzati. Le istruzioni fornite consentono di preparare 4 dischi principali:

- GRAFICA IN ALTA RISOLUZIONE
  Disegnare in modo GRAPHIC 1; salvare i disegni sul disco; tagliare parti di un disegno e incollarle in un altro; eseguire varie operazioni atte a creare disegni astratti.
- GRAFICA IN BASSA RISOLUZIONE
  Le stesse possibilità in modo GRAPHIC 3 (multicolor).
- GRAFICA AZIENDALE
  Diagrammi a barre, istogrammi, curve lineari, grafici a torta, diagrammi di distribuzione (scattergram); convertire dati alfanumerici in rappresentazioni grafiche; salvare i dati in file sequenziali oppure salvare il grafico stesso in forma di bit-map. Possibilità di usare i programmi del Lato 1 e 2 per inserire titoli addizionali o elementi personalizzati di disegno.
- GALLERIA
  Consente di riempire un disco qualsiasi di disegni bitmappati, e poi di eliminare tutti i programmi ad eccezione di LETTORE, che a sua volta consente di presentare ciascun disegno sullo schermo per un periodo di tempo regolabile da un secondo fino a diverse ore, dopo aver pre-

sentato una schermata contenente il titolo, il nome dell'autore e fino a 5 righe di commento.

Questi programmi non devono essere scambiati per programmi di grafica professionale: se il loro scopo fosse stato semplicemente quello di permettere all'utente di disegnare, molte operazioni avrebbero potuto essere rese più automatiche, nascondendo totalmente le funzioni del BASIC. Inoltre, i programmi potrebbero essere resi molto più veloci usando un compilatore per produrre una versione finale in linguaggio macchina. Per questa importante possibilità si veda il Capitolo 2.

1

La tastiera

Il C-128 e il C-128D, considerati come oggetti, sono decisamente eleganti. La tastiera, completamente diversa per quanto riguarda l'aspetto dalle tastiere Commodore classiche (VIC-20, C-64, C-16, ecc.), è di colore bianco panna o beige chiarissimo. È larga e piatta, con tasti disposti piuttosto geometricamente.
I tasti sono molleggiati e di dimensioni giuste rispetto alle dita. Non fanno l'irritante "clic-clic" di certe tastiere, ma si sente il suono del loro movimento, e il dito si accorge se il tasto è stato premuto. Per chi è abituato a usare macchine da scrivere, terminali e altre tastiere, l'impressione è decisamente buona. La disposizione è del tipo QWERTY.
In questo capitolo si esamineranno soltanto i tasti che non esistono sugli altri calcolatori Commodore, o che hanno assunto funzioni diverse.

1.1 Il tastierino numerico

Sul lato destro c'è un tastierino numerico con i tasti da 1 a 0, +, − , . e ENTER. Nell'uso normale ENTER è uguale a RETURN, ma certi programmi commerciali distinguono tra i due per ottenere risultati diversi. Una piccola differenza è la seguente: in un programma in BASIC, non è possibile interrompere un INPUT usando STOP. Premendo simultaneamente STOP e ENTER il programma si interrompe, mentre STOP e RETURN non hanno questo effetto.
I tasti numerici hanno lo stesso effetto degli altri tasti numerici posti in fila orizzontale. A differenza da questi non sono influenzati da SHIFT.

Il tastierino numerico è particolarmente utile quando si devono inserire lunghe serie di dati numerici.

1.2 ESCAPE

Questo è un tasto per funzioni speciali: battendolo non compare nessun carattere sullo schermo. In un programma si ottiene la stessa funzione con PRINT CHR$(27). Il tasto ESCAPE serve per introdurre comandi speciali: in modo C-128 ha funzioni fisse quando è seguito da un altro tasto, offrendo così funzioni particolarmente utili nella compilazione di programmi in BASIC. Tra queste:

ESC-O disabilita il "modo virgolette" che inizia dopo aver battuto SHIFT-2 o INST e provoca la comparsa di caratteri inversi corrispondenti a colori, frecce, eccetera.

ESC-Q cancella tutto dal cursore fino alla fine della linea di programma corrente.

ESC-P cancella all'indietro dal cursore fino all'inizio della linea di programma.

ESC-@ cancella tutto il resto dello schermo a partire dal cursore.

ESC-A abilita l'auto-insert (modo in cui, quando si batte un carattere sopra un altro, quest'ultimo non viene cancellato, ma si sposta verso destra). Questo modo termina con ESC-C. L'auto-insert rallenta notevolmente le operazioni di stampa sullo schermo, e questo può essere fastidioso. Si può però sfruttare questa lentezza in un modo forse non previsto dalla Commodore ma non per questo meno utile, in alternativa a NO SCROLL, per rallentare il LIST. Anche la pressione del tasto COMMODORE (il tasto più in basso a sinistra) rallenta il listato, ma ESC-A è più utile.

ESC-C termina il modo auto-insert iniziato con ESC-A.

ESC-J/K va, rispettivamente, all'inizio e alla fine della riga corrente.

ESC-D cancella la riga corrente.

ESC-I inserisce una riga (vuota) tra due righe di programma.

ESC-Y riporta il tasto TAB al valore iniziale (di default: pari a 8 spazi).

ESC-M modifica lo "scroll" quando si lista un programma. Non ha la stessa funzione del tasto NO SCROLL: mentre quest'ultimo, come CONTROL-S, arresta il programma che "scrolla", ESC-M fa sì che, quando è piena l'ultima riga dello schermo, si torna alla prima invece di muovere verso l'alto le righe già visualizzate. È utile, anche se produce un effetto un po' sconcertante, e bisogna abituarvisi. ESC-L lo termina.

ESC-V scroll verso l'alto.

ESC-W scroll verso il basso.

ESC-E ferma il lampeggio del cursore.
ESC-F abilita il lampeggio del cursore.
ESC-B definisce l'angolo inferiore destro di una finestra sullo schermo.
ESC-T definisce l'angolo superiore sinistro di una finestra.
ESC-X cambia tra 40 e 80 colonne. Mentre il tasto 40/80 DISPLAY funziona principalmente al momento dell'accensione, ESC-X funziona sempre.

Solo per lo schermo a 80 colonne:

ESC-U cambia il cursore in un carattere di sottolineatura (_, underscore).
ESC-S torna al cursore normale.
ESC-R cambia a campo inverso (reverse).
ESC-N torna a campo normale.

Il carattere corrispondente a ESC ha il codice ASCII CHR$(27), e tramite l'uso di questo carattere le funzioni elencate sopra sono disponibili anche tramite programma. Per esempio

```
10 print chr$(27);"r"
```

produce lo stesso effetto (campo inverso) che si ottiene premendo in modo diretto ESC-R.
Molte possibilità ottenibili con ESC e un altro carattere sono ottenibili anche con CONTROL e un altro tasto, ma in diversi casi CONTROL dà la possibilità di ottenere un effetto ma non quella di annullarlo.
Le finestre (ESC-B e ESC-T) sono più comodamente ottenibili in programma tramite il comando WINDOW (si veda il Capitolo 9).

1.3 TAB

Questo tasto normalmente inserisce 8 spazi. Si può stabilire una posizione di tabulazione in un punto qualsiasi della riga premendo CONTROL-X. ESC-Y, invece, ristabilisce il default. In "modo virgolette", TAB produce caratteri inversi (come i codici-colore), e la presenza di questo carattere in una stringa equivale a 8 spazi.

1.4 ALT

Il manuale del 128 lo definisce come uno dei tasti che funzionano solo in modo 128 ma non dà una spiegazione precisa. Premendo ALT e un altro tasto, se quest'ultimo è stato ridefinito da un programma, si ottiene un risultato speciale. Il sistema operativo CP/M usa questo tasto per una funzione simile.

1.5 CAPS LOCK

Un tasto utilissimo. È un altro "toggle", come SHIFT LOCK e 40/80 DISPLAY. Quando si preme SHIFT LOCK si ottengono lettere maiuscole, ma anche il carattere "shiftato" degli altri tasti. Se battiamo

a 1 b 2 c 3 d 4 . , : ;

e poi gli stessi tasti con SHIFT LOCK, otteniamo:

A ! B " C # D $ < > [ ]

Questo può essere scomodo, se vogliamo scrivere un testo tutto maiuscolo ma con numeri e segni di interpunzione, oppure se il programma che stiamo usando richiede l'immissione di lunghe serie di dati numerici. CAPS LOCK, se premiamo di nuovo gli stessi tasti, fornisce:

A 1 B 2 C 3 D 4 . , : ;

Per ottenere invece i segni ! " # $ % ecc., basta premere SHIFT come al solito.
Purtroppo, questo tasto è vittima di un difetto vistoso, sebbene non grave, del sistema operativo nelle versioni del C-128 e del C-128D in vendita in Italia, ad eccezione dei modelli con tastiera italiana (si veda più avanti). La Q non è influenzata da questo tasto. Per ottenere la Q maiuscola è sempre necessario premere SHIFT oppure SHIFT LOCK. Perciò, se si devono inserire numeri e parole in tutto maiuscolo, è consigliabile servirsi di SHIFT LOCK e del tastierino numerico, anche se in questo caso i tasti CLR/HOME e INST/DEL saranno shiftati.

1.6 La tastiera italiana

Per chi aspira principalmente a scrivere programmi, la disposizione della tastiera non ha certo molta importanza. Ma il C-128 può essere usato come sistema di videoscrittura, usando lo schermo a 80 colonne e un buon word processor. Se immaginiamo un dattilografo abituato a scrivere con una macchina normale, è chiaro che la disposizione dei tasti assume una certa importanza. La tastiera italiana delle macchine per scrivere differisce da quella anglosassone per la posizione (QZERTY...) di alcuni caratteri, e per la presenza delle vocali con accento (è, à, ù, ecc.). La diversa posizione dei tasti è dovuta al fatto che certe lettere, frequenti in inglese, servono solo raramente in italiano. Così per esempio la Z, più comune in italiano, viene ad occupare un posto più comodo. Come è descritto nel manuale, l'uso di CAPS LOCK consente anche di ottenere questa disposizione dei tasti. Quando si seleziona la tastiera italiana, entrano in vigore i caratteri che, su certi tasti, sono indicati in grigio. Così appaiono sullo schermo anche lettere accentate, e vengono inviati alla stampante i codici ASCII corrispondenti.

1.7 HELP

Se si verifica un errore durante l'esecuzione di un programma in BASIC, quando si preme HELP appare sullo schermo la riga di programma, con i caratteri invertiti, cioè in negativo, a partire dal punto in cui il calcolatore ha rilevato l'errore. Sullo schermo a 80 colonne la riga è invece sottolineata a partire dallo stesso punto. HELP è un comando del BASIC 7.0, e premere questo tasto è esattamente come battere HELP RETURN. È un tasto funzione, e come tale può essere programmato (si veda più avanti).

1.8 RUN

Mentre sul C-64 la pressione di questo tasto carica automaticamente il primo programma su nastro, sul C-128 carica ed esegue il primo programma sul disco. Come HELP, è un tasto funzione specializzato e può essere riprogrammato.

1.9 LINE FEED

Premendo questo tasto il cursore va giù di una riga, come quando si preme la freccia verso il basso, oppure RETURN. Quest'ultimo tasto, che dà il codice ASCII CHR$(13), termina una riga e porta il cursore (o la stampante) alla prima colonna della riga successiva. LINE FEED, invece, va semplicemente giù di una riga. Corrisponde a CHR$(11). È molto utile con la stampante.

1.10 40/80 DISPLAY

Si veda il paragrafo 9.1 "Lo schermo a 80 colonne".

1.11 NO SCROLL

"Scrolling" è il termine che si usa per indicare righe che scorrono rapidamente sullo schermo, di regola dal basso verso l'alto, come quando si lista un programma in BASIC. Per fermare lo scroll si può premere i tasti CONTROL e S contemporaneamente, ma bisogna essere velocissimi. NO SCROLL, essendo un tasto solo, è più veloce. In entrambi i casi, premere un tasto qualsiasi, compreso NO SCROLL, per rimettere in moto le righe. Si veda però anche l'osservazione precedente relativa ad ESC-A.

1.12 Le frecce

Servono per portare il cursore in qualunque parte sullo schermo. In BASIC, all'interno di una stringa (cioè tra virgolette) stampano speciali caratteri invertiti che hanno lo stesso effetto quando si stampa la stringa. Le frecce nella parte alta della tastiera funzionano in modo C-128 e in modo CP/M.
Immediatamente sotto il tasto RETURN troviamo altre quattro frecce su due tasti (da usare con e senza SHIFT). In modo C-64 funzionano solo questi, mentre in modo CP/M non funzionano affatto. Né gli uni né gli altri sono molto comodi. Vari programmi grafici sul Lato 2 del nostro disco usano invece i tasti del tastierino numerico per spostare il cursore (8 e 2 per movimenti verticali, 4 e 6 orizzontali, 7-9-1-3 diagonali).

1.13 I tasti funzione

Questi quattro tasti, con e senza SHIFT, danno otto possibilità di scrivere, con la pressione di un solo tasto, stringhe che possono essere parole o numeri qualsiasi, o anche comandi completi per il calcolatore. La formula per programmare un tasto funzione è:

KEY*n*, "*stringa*"

e può essere usata anche all'interno di un programma in BASIC. Il programma KEY, sul Lato 1 del nostro disco, ne dà spiegazioni complete. Consente inoltre di riprogrammare i tasti funzione specializzati RUN e HELP. KEY è inoltre un comando del BASIC 7.0. Battere la parola chiave KEY senza argomento produce sullo schermo la lista precisa del contenuto delle 8 funzioni. Eccone la riproduzione, ottenuta con il seguente programmino:

```
1 dopen#1, "key.list",w
10 cmd1, "key.list"
20 key
30 print#1: dclose
```

```
key.list
key 1,"graphic"
key 2,"dload" + chr$(34)
key 3,"directory" + chr$(13)
key 4,"scnclr" + chr$(13)
key 5,"dsave" + chr$(34)
key 6,"run" + chr$(13)
key 7,"list" + chr$(13)
key 8,"monitor" + chr$(13)
```

KEY 1 risparmia la fatica di battere "GRAPHIC" per esteso.
KEY 2 DLOAD, il comando per caricare un programma da dischetto. Si aggiunge il nome del programma e si batte RETURN. Le virgolette d'apertura [CHR$(34)] sono comprese.
KEY 3 battere questo tasto equivale a scrivere DIRECTORY e poi battere RETURN (codice ASCII 13).
KEY 4 svuota lo schermo senza bisogno di battere altri tasti. [v. Dizionario].
KEY 5 DSAVE, come DLOAD: salva un programma su disco.
KEY 6 "RUN"+CHR$(13) esegue il programma in memoria.
KEY 7 LIST, mostra il listato del programma in memoria.
KEY 8 MONITOR dà accesso al "monitor" che consente di programmare in codice macchina.

Si noti come alcuni comandi sono seguiti da CHR$(13), che equivale a premere RETURN. La pressione di questi tasti ha un effetto immediato. Laddove manca CHR$(13) è necessario completare il comando, aggiungendo per esempio il nome del programma, e poi premere RETURN.
I tasti funzione possono essere programmati in qualsiasi momento, anche più volte all'interno di uno stesso programma, o direttamente da tastiera. Definirli da tastiera è comodo quando, per esempio, si deve scrivere una istruzione numerose volte in un programma (come "Premere un tasto": GETKEY A$). Un modo facilissimo di riprogrammare i tasti è di battere KEY per produrre la lista riprodotta più sopra e poi correggere le varie righe, premendo RETURN non appena la riga è come desiderata. Sarà poi necessario battere KEY di nuovo per correggere il tasto successivo (perché la pressione di RETURN produce il prompt "ready" sotto la riga appena corretta).
È evidente che la scelta delle funzioni di default, cioè quelle esistenti al momento dell'accensione, è arbitraria: chi ha creato il sistema operativo ha scelto comandi che possono essere utili alla maggior parte degli utenti. Chi ha esigenze diverse può facilmente creare un proprio "profilo utente" che cambi molti parametri di default.

I seguenti tre tasti non sono nuovi, ma vale la pena di ricordarne l'uso.

1.14 CONTROL

Questo tasto non è autonomo: deve essere premuto sempre insieme a un altro tasto. In modo diretto CTRL-(altro tasto) ha immediatamente l'effetto voluto (CTRL-3 colora il cursore di rosso, CTRL-G suona il "campanello", ecc.). In "modo virgolette" la pressione di CTRL con un altro tasto produce un carattere in negativo (non necessariamente il carattere indicato sul tasto premuto). Durante l'esecuzione del programma questo carattere produce lo stesso effetto che si ha con CTRL e quel carattere in modo diretto.
Anche se non tutti sono utili, molti codici di controllo sono indispensabili. L'Appendice I della System Guide fornisce un elenco di questi codici, che presentano alcune differenze per i modi 64 e 128.

1.15 COMMODORE

È il tasto più in basso a sinistra. Ha funzioni simili a CTRL. Dà per esempio i colori numerati da 9 a 16 (si veda la voce COLOR nel Dizionario del BASIC) quando viene premuto insieme ai numeri 1-8. Il suo uso più importante è come alternativa al tasto SHIFT.

Nel modo default (maiuscole e caratteri grafici) in cui si trova il sistema al momento dell'accensione, il tasto COMMODORE consente di ottenere il carattere grafico raffigurato più a sinistra sulla maggior parte dei tasti, mentre SHIFT consente di ottenere quello più a destra.
La pressione simultanea di COMMODORE e SHIFT consente di passare dal modo "maiuscolo e caratteri grafici" al modo "maiuscolo e minuscolo" (e viceversa). Una novità importante è che, mentre in modo 40 colonne la pressione di COMMODORE-SHIFT cambia anche i caratteri già visualizzati sullo schermo, in modo 80 cambia solo i caratteri battuti successivamente. Ciò significa che i caratteri disponibili sullo schermo ad 80 colonne ammontano a 512. Si vedano il capitolo relativo e il programma 80-COL sul disco (Lato 1).

1.16 RESTORE

Un altro tasto che non ha effetto se premuto da solo. Premuto insieme a STOP ferma l'esecuzione di un programma. Lo schermo torna ai colori di default, il programma si ferma, ma normalmente resta intatto. In genere questa è la cosa da fare quando si è perduto il controllo del calcolatore. Se non funziona si rischia di perdere anche il programma. Infatti non resta che premere il tasto RESET.

1.17 RESET

È il tasto piccolo sul lato destro del calcolatore, rientrato per evitare una pressione accidentale. La pressione di questo tasto riporta il sistema a uno stato simile a quello in cui si trova all'accensione "a freddo".
Nel caso del blocco del sistema dovuto a un errore di programma può capitare in certi casi che, fallito un tentativo con STOP e RESTORE (v. sopra), se non si dispone di una copia del programma e si rischia di perderlo, si può salvare la situazione premendo STOP e RESET insieme. Ciò funziona quasi sempre. Bisogna tenere premuto STOP per alcuni secondi. Quando è terminata l'operazione ci si trova nel Monitor. Da questa situazione si esce battendo X e poi RETURN, e il programma dovrebbe risultare intatto. E se tutto ciò non avesse effetto, inserire in modo diretto:

```
BANK 15: POKE(PEEK(45)+PEEK(46)*256),1: SYS24293
```

Vale comunque la pena di sottolineare che queste sono misure d'emergenza: un buon programmatore non cerca mai di eseguire un programma senza averlo già salvato e verificato.

2

Programmazione

Questo capitolo si occupa della struttura dei programmi. In certi casi fa riferimento ad argomenti trattati più avanti nel testo, ma non è indispensabile conoscerli ai fini di questa discussione. Sono comunque reperibili tramite l'indice analitico e il Dizionario.
Con il BASIC 7.0, come abbiamo già visto, è possibile evitare l'uso del linguaggio macchina, anche sotto forma di PEEK e POKE, per sopperire all'assenza di comandi specifici del BASIC. Nessuno dei programmi sul disco che accompagna questo volume usa PEEK o POKE per compiere operazioni non previste dal BASIC. Sul Lato 2 POKE 4184,128 viene usato per dire al programma chiamato di lavorare sul grafico già in memoria, e quindi di non pulire lo schermo. Questa, però, è una mera registrazione di un dato in una posizione di memoria normalmente inutilizzata; un'operazione insolita che nessun linguaggio di programmazione potrebbe prevedere. È quasi sempre possibile scrivere tutto il programma in BASIC 7.0, ma in certi casi il programma può risultare troppo lento. Se è vero che la ricerca della velocità a tutti i costi è un errore, è anche vero che un programma come ALFASORT (Lato 1), se deve alfabetizzare molte stringhe (per esempio un migliaio), è molto lento.
ALFASORT è presente sul disco solo in BASIC. Usa la procedura più semplice (sort "a bolle"), e in realtà la soluzione migliore sarebbe di riscriverlo in maniera più efficiente. Ma anche così risulterebbe lento, e la soluzione globale consisterebbe, dopo averlo reso più efficiente, nella compilazione di una versione in linguaggio macchina.
Fortunatamente, esiste la possibilità di ottenere un programma in linguaggio macchina senza conoscere quest'ultimo. Certi linguaggi ad alto livello (come il FORTRAN, il Pascal, o il linguaggio C) funzionano sempre

in questo modo. Si scrive un file (codice sorgente) usando uno speciale editor oppure un normale word processor: questo file viene usato dal compilatore per creare un programma (codice oggetto), in linguaggio macchina oppure in un codice speciale chiamato p-code (pseudo-codice, simile al codice macchina ma più compatto).

2.1 Interpreti e compilatori

Il BASIC e certi altri linguaggi usano invece un interprete. Anche questo traduce il testo "umano" del BASIC in codice macchina, ma esegue i comandi immediatamente, tutte le volte che si dà il RUN al programma. Mentre si scrive il programma, l'interprete è molto utile, perché consente di provare il programma, o una sua parte, immediatamente, tutte le volte che si desidera. Poiché però l'interprete deve eseguire il proprio lavoro in tempo reale mentre il programma "gira", il BASIC può essere lento.
Un compilatore è un interprete speciale che registra il proprio lavoro in un file oggetto che diventa il programma vero e proprio. Un esempio di questo genere è fornito sul nostro disco dal programma DIRTUT. Sul Lato 1 esiste la versione in BASIC (codice sorgente), mentre il Lato 2 comprende lo stesso programma compilato in p-code: soltanto i titoli presentati all'inizio sono diversi. Il programma legge il direttorio del disco ed esegue varie operazioni (presenta il direttorio in vari modi ottenendo fino a 80 nomi di file sullo schermo a 80 colonne, esegue ricerche, ecc.). La versione compilata sul Lato 2 è circa 5 volte più veloce, anche se purtroppo le operazioni di lettura del disco non vengono accelerate. Naturalmente non si può listare questa versione per capirne il funzionamento, ma il codice sorgente è disponibile sull'altro lato.
Il compilatore usato è BASIC 128 della Data Becker, ma esistono altri compilatori per il C-128, per il BASIC 7.0 e per altri linguaggi. BASIC 128 consente la compilazione in p-code, in LM o in una combinazione dei due. Sono state compilate versioni in p-code e in LM: la versione in p-code è stata preferita perché risultava più breve e non era molto più lenta.
Per fare un solo esempio, la velocità con cui il calcolatore esegue un GOSUB, quando il programma è in codice sorgente, dipende dalla distanza (numero di righe) tra il GOSUB stesso e il numero di riga specificato; e analogamente la velocità dell'esecuzione del RETURN dipende dal numero delle righe intercorrenti tra l'inizio del programma e il GOSUB di partenza. Questo è perché l'interprete deve calcolare il salto ogni volta. Il compilatore, invece, esegue il calcolo una sola volta e ne inserisce il risultato permanentemente nel programma compilato. Di conseguenza, un GOSUB è immediato in tutti i casi.

In genere, per usare un compilatore, non è necessario conoscere il codice macchina. Con BASIC 128 non è nemmeno indispensabile sapere che cosa sia.

2.2 Struttura del programma

Con il C-128, che può accettare programmi molto lunghi, assumono nuova importanza la struttura e la leggibilità del programma, anche quando non esiste l'esigenza di usare il compilatore. Dividere il programma in blocchi, scrivere righe brevi senza mischiare operazioni di tipo diverso nella stessa riga, e così via, sono buone abitudini che consentono una maggiore sistematicità nella stesura del programma e rendono più facile la lettura e il controllo, anche se non si vuole usare un compilatore. Alcune di queste pratiche possono però rallentare ulteriormente il programma: in questi casi l'uso del compilatore consente di disporre di un codice sorgente leggibile e di un codice oggetto efficiente.
Diciamo subito che non costituisce reato creare un programma che non abbia una struttura chiara e che non sia facilmente leggibile, se funziona. È però frustrante e dannoso non riuscire a rivedere un programma perché la sua struttura è ingarbugliata, ed è umiliante riprendere in mano un nostro programma a distanza di tempo e trovare che il calcolatore lo capisce, mentre noi non ci riusciamo più.
Un primo passo è scrivere il programma in modo che si capisca il rapporto fra le righe e fra le varie routine:

```
100 do until a$ = "!"
105 :   print "Premere un tasto"
110 :   getkey a$
120 :   if a$ = "a" then begin
125 :       print "E´ stato premuto il tasto";
130 :       print a$
140 :       for t = 1 to 5
150 :           print t
160 :           print "Questo e´ solo un esempio"
165 :           print "che viene stampato 5 volte"
    :           print "per illustrare i loop annidati"
170 :       next t
175 :       print "Con questo termina la routine FOR... TO"
180 :   bend
185 :   print "E qui termina la serie BEGIN/BEND"
190 loop
200 print "Chiuso anche il DO... LOOP: il programma termina"
```

Questo programma contiene 3 routine annidate con una struttura che, senza influenzarne il funzionamento, rende facile capire i meccanismi del

programma. I due punti servono per conservare gli spazi all'inizio delle righe, che altrimenti verrebbero ignorati dall'interprete BASIC del C-128. Tutte le righe a uno stesso livello di struttura si trovano alla stessa distanza dal margine sinistro. Questo consente anche di accertare con un colpo d'occhio che ciascuno dei loop venga chiuso, e che si chiuda nell'ordine giusto. Con BEGIN...BEND e DO...LOOP potrebbe altrimenti essere facile dimenticare una chiusura. Se ciò accade il programma può anche funzionare per un certo periodo di tempo: se però lo stesso loop viene aperto più volte, allora si ha un loop dentro un loop dentro un loop... fino al momento in cui l'intera memoria delle variabili nel Banco 1 è occupata, e il programma blocca il sistema. Possiamo anche inserire una riga vuota:

```
210 :
```

prima di iniziare un altro blocco del programma. Questo rende ancora più facile capire (e ricordare) la suddivisione funzionale delle parti del programma. Se si usa un compilatore, questo provvede all'eliminazione di tutti questi spazi (come anche delle REM e dei numeri di riga) che perciò non allungano il file di codice oggetto.
Chi ha mai scritto programmi per calcolatori dotati di poca memoria (come il VIC-20 o il C-16: quest'ultimo in un programma grafico ha circa 2 K per il programma vero e proprio) avrà avuto anche l'abitudine di scrivere righe di programma della lunghezza massima possibile, addirittura per ridurre il numero delle righe, perché ogni numero occupa memoria. Con costrizioni del genere, la struttura del programma non può essere considerata. Con il C-64, e ancor più con il C-128, il numero di comandi in una riga dovrebbe dipendere dal contenuto della riga:

```
1000 print"ssS"; chr$(14): color0,16: color4,1:
     color5,1: color6,16:
     bload"key.bin",p4096
```

Questa, per esempio, del programma KEY, potrebbe essere riscritta in 7 righe separate. Poiché è una riga di semplice routine (pulisce lo schermo e ne definisce i colori, poi carica il file "key.bin" che ripristina il contenuto dei tasti funzione ai valori iniziali) sembra logico farne una riga unica, per occupare meno spazio sullo schermo e sulla carta.

```
1130 dl$ = dl$ + chr$(13): d2$ = d2$ + chr$(13)
1140 poke 4104,len(dl$): poke 4105,len(d2$)
1150 for t = 1 to len(dl$) + len(d2$)
1160 : x = asc(mid$(dl$ + d2$,t,1))
1170 : if x = 95 then x=13
1180 : poke 4158 + t,x
1190 next
```

Qui, invece, vediamo operazioni specifiche del programma: la prima riga accoglie due definizioni parallele, ed è quindi sembrato logico metterle insieme, mentre la seconda effettua due operazioni esattamente corrispondenti alle due definizioni.
Il loop tra 1150 e 1190, invece, potrebbe essere contenuto in una sola riga, tenendo presente che il C-128 accetta righe di programma fino a 160 caratteri, ma è rigorosamente suddiviso secondo le operazioni. Si è scelto per l'esempio uno dei pochi programmi sul disco che usano PEEK e POKE, proprio perché in un caso del genere diventa più che mai importante la chiarezza. Per chi desidera capire il funzionamento della routine, gli indirizzi 4104 e 4105 contengono la lunghezza delle stringhe contenute nei tasti RUN e HELP, e le stringhe (quando i primi 8 tasti funzione contengono le stringhe di default) vengono inserite a partire da 4159. La routine inserisce prima le lunghezze, poi le stringhe.
In ogni caso risulta chiaro che la struttura illustrata è molto più leggibile di una sola riga. Si potrebbe anche inserire una coppia di REM:

```
1125:
1126 rem: 4104 = lunghezza stringa run:
          4105 = lunghezza stringa help
          4159 = inizio stringhe funzioni
1128:
```

Queste REM sono inserite prima del blocco di programma al quale si riferiscono e sono separate dal blocco precedente e da quello che segue da due righe vuote, in modo da non interferire con la lettura delle righe operative. Più comunemente una REM prima di un blocco potrebbe essere, per esempio

490 REM: disegnare cerchio con raggio x,y, colore f

I programmi sul disco che accompagna questo volume fanno un uso limitatissimo di REM, in parte per risparmiare spazio sul disco e in parte per una questione di stile personale dell'autore.
Particolarmente se il programma in BASIC sarà compilato e se non esistono problemi di spazio sul disco, si possono inserire tutte le REM che si desidera. Infatti il compilatore le ignora completamente e quindi non aggiungono nulla alla lunghezza del programma, mentre il codice sorgente, che può essere conservato su un altro disco, resterà per sempre comprensibile.
La struttura a blocchi è particolarmente utile quando "si usa" il compilatore, anche perché è possibile fare una compilazione "mista", compilando certe parti in p-code e certe altre in linguaggio macchina puro. In questo caso è necessario che un intero blocco sia nello stesso codice e che si

evitino i salti dall'interno di un blocco in un tipo di codice a un blocco in un altro tipo di codice. Seguire in ogni caso il manuale del compilatore.

2.3 Overlay

Il fatto che il C-128 può riservare ai programmi fino a 64 K di memoria non significa che sia una buona idea creare programmi troppo lunghi. Sul nostro disco nessuno dei programmi in BASIC supera circa 12 K: inoltre, i programmi più lunghi (come per esempio JOYDRAW) devono poter saltare rapidamente da un blocco all'altro in qualsiasi momento. Perciò la parte principale è un programma unico. Nonostante questo, per salvare i disegni il programma salta a SALVAGRAFICI, programma che serve anche a DRAW e a MULTIDRAW, e in questo caso si risparmia quindi anche spazio sul disco.
La situazione cambierebbe se dovessimo creare, per esempio, un programma di contabilità. Questo avrebbe delle suddivisioni logiche molto precise: incassi, spese, crediti verso clienti, debiti verso fornitori, IVA, inventario, e così via. L'utente userebbe il programma per operazioni specifiche, e il tempo di permanenza in ciascuno dei vari "ambienti" sarebbe ben determinato. Sembrerebbe quindi più logico creare una serie di programmi in "overlay": cioè con un menu centrale e una serie di programmi specifici per ciascuna operazione contabile. Overlay (sovrapposizione) significa in questo caso che tutti i programmi si caricano con DLOAD, indirettamente via menu o direttamente l'uno dall'altro, e non con RUN come la maggior parte dei programmi sul nostro disco. In questo modo, infatti, si ha una "partenza a caldo" (warm start): i valori inseriti nelle variabili restano intatti, e le operazioni possono continuare senza soluzione di continuità. Nel caso di una serie di programmi di questo genere l'uso di un compilatore consente generalmente un overlay molto economico anche di spazio sul disco, poiché il run-time module può essere unico. Il run-time module, lungo tipicamente 8-10 K, è la serie di informazioni necessarie per l'operazione di tutte le routine di uno o più programmi compilati, e naturalmente si può organizzare la struttura in modo che questo debba essere caricato una sola volta all'inizio della sessione di lavoro. Tutti gli altri programmi possono esserne privi, e quindi risulteranno anche più brevi dei programmi originali in BASIC. Ciò riduce sia il tempo di caricamento dei singoli programmi, sia lo spazio globalmente occupato sul disco.
Il vantaggio principale per un sistema di programmi contabili o simili è però la facilità di aggiornamento dei singoli programmi. Il Ministero delle Finanze, come si sa, ama cambiare le leggi fiscali continuamente: un'azienda cambia la struttura dei prezzi o chiede al calcolatore di pre-

sentare i dati in un altro modo. Se il programma è suddiviso in moduli, e se ciascun modulo ha una sua struttura ben precisa, diventa molto più facile sia ricercare gli errori di programma, sia ristrutturare ciascuna singola parte, senza il rischio di introdurre errori nelle parti che non richiedono modifiche. Ricordiamo però che la velocità di un programma non è una virtù in sé. In altre parole, l'uso di un compilatore può non rappresentare un vantaggio reale. Mentre un'azienda che vende al dettaglio può avere l'esigenza di registrare centinaia di operazioni al giorno, un libero professionista può avere la necessità di registrare soltanto un'operazione al giorno. In quest'ultimo caso può convenire mantenere l'intero pacchetto di programmi in BASIC, perché nessuno si renderà conto della relativa lentezza del programma.

2.4 GOSUB e GOTO

Ritorniamo al programma KEY per esaminare l'uso di GOSUB e di GOTO. Teniamo presente che ogni GOSUB tende a rendere meno comprensibile il programma per chi ne legge il listato, particolarmente se lo legge sullo schermo. KEY avrebbe potuto contenere un GOSUB del seguente tipo:

```
1050 gosub 30000:rem  routine line input
1060 dl$ = x$: x$ = "": a$ = ""
......
30000 do until a$ = chr$(13)
30010 :  get a$: print a$;: x$ = x$ + a$
30020 loop
30030 return
```

Nel caso specifico, una routine del genere serve soltanto due volte e quindi sono state usate due routine senza eseguire alcun salto per mantenere la massima leggibilità del programma; se invece servisse molte volte, sarebbe utile. Usa due variabili locali (che infatti vengono azzerate dopo il GOSUB), ed è quindi disponibile per accettare qualsiasi stringa e non soltanto dl$. È una routine che supera l'assenza di un LINE INPUT nel BASIC 7.0, accettando anche i caratteri virgola e due punti, che un INPUT normale non accetta (tronca la stringa e dà il messaggio EXTRA IGNORED).
Non è sempre possibile generalizzare una routine fino a questo punto, naturalmente, ma sarà sempre una buona regola cercare di farlo, in modo che possa essere usata in tutto il programma. Nel BASIC 2.0, può capitare che una routine da eseguire dopo un IF occupi più di una riga. Per

evitare una serie di righe tutte con IF X=3..., esistono due soluzioni: una è un salto con

```
50 if x<>3 then 100
60 print x... ecc.
```

e l'altra, spesso adoperata, è di creare una subroutine ad hoc e collocarla in qualche altro punto del programma. Come si è visto nel primo esempio di questo capitolo, BEGIN... BEND in BASIC 7.0 rende inutili questi sotterfugi, poiché qualsiasi numero di righe dopo IF X=3 THEN BEGIN, fino a BEND, verranno eseguite soltanto se x=4.
Valgono considerazioni analoghe anche con GOTO. Se non è necessario eseguire un salto, sarà meglio evitarlo, anche se fosse necessaria qualche riga di programma in più: un salto utile condiziona tipicamente tutto il funzionamento successivo del programma, come nei menu sul disco, con

```
ON ASC(X$)-64 GOTO ...
```

in cui ciascuno dei GOTO carica un programma diverso. Un salto tipicamente sbagliato si riscontra quando durante la stesura del programma si sono consumati tutti i numeri e, non potendo aggiungere una riga tra il 994 e il 995, il programmatore attacca in coda un GOTO 10000, aggiunge le righe che vuole e le fa terminare con GOTO 995. Con il C-64 ciò è giustificabile a causa della fatica necessaria per rinumerare un programma (giustificabile ma non giusto). Poiché il C-128 dispone del comando RENUMBER, il programmatore può usare RENUMBER, oppure RENUMBER 1000,5,995.
Il primo rinumera tutto il programma con intervalli di 10, mentre il secondo rinumera, a partire da 1000 e con intervalli di 5, soltanto le righe successive alla 994.
Abbiamo fatto alcune osservazioni relative allo stile nella programmazione visto dal punto di vista del programmatore, nell'intento di sensibilizzare il lettore alla questione più che di offrirne una discussione completa. Da un punto di vista completamente diverso, è opportuno che il programmatore abbia un proprio stile nel rapporto con l'utente del programma.

2.5 Il rapporto con l'utente

Spesso, infatti, la differenza tra un programma buono e uno scadente, a parità di funzioni e di possibilità, consiste nell'attenzione e nella considerazione dovute all'utente: colori e disposizione del testo possono stancare

o distrarre gli occhi: messaggi troppo succinti possono creare confusione, e così via.
Ciò riguarda in particolare due aspetti:

- Il modo di chiedere dati all'utente. Ogni volta che si richiede un dato di un determinato tipo, sarà opportuno chiederlo con lo stesso tipo di messaggio: con o senza il "ping" del campanello, con o senza un messaggio scritto, con INPUT o GETKEY, ma possibilmente sempre allo stesso modo, con caratteri dello stesso colore, e così via. Si ridurrà così il rischio di creare confusioni. Sul nostro disco si è seguita una regola del genere, anche se con delle variazioni per illustrare le diverse possibilità. A questo punto è necessario decidere se dare del tu all'utente ("Premi un tasto"), oppure evitare di farlo ("Premere un tasto"). È solo un dettaglio, ma contribuisce allo stile del programma.
- Lo stile della presentazione dei dati sullo schermo. Il C-128, particolarmente sullo schermo a 80 colonne, è capace di creare schermi assai eleganti ed efficaci, se il programmatore si prende il disturbo di studiarne gli aspetti artistici. L'uso di finestre (anche sullo schermo a 40 colonne) consente di mantenere una impaginazione standardizzata, e i 16 colori disponibili consentono di evidenziare le scritte in modi sempre diversi e attraenti. A seconda del tipo di programma si possono alternare i colori oppure usare determinati colori per determinati tipi di dati. Se un programma fa parte di una serie, sarà opportuno considerare la serie globale da questo punto di vista. Per esempio i programmi sul Lato 1 del nostro disco hanno quasi sempre lo stesso schema dei colori (che è quello che, dopo molti esperimenti, sembrava il più adatto sia per le 40 che per le 80 colonne), e molti di questi programmi hanno un titolo corrente nella prima riga dello schermo per aiutare il lettore a capire quale programma sta usando. Potremmo, per tornare al nostro gruppo di programmi contabili in overlay, decidere di usare colori diversi per ciascun singolo programma. I nostri programmi sono molto numerosi, infatti, e perciò il titolo corrente è sembrato la soluzione migliore. Il gruppo di programmi contabili sarebbe meno numeroso e ciascun programma verrebbe usato più frequentemente: dovrebbe quindi bastare una piccola variazione nella grafica per evitare che l'utente perda l'orientamento.

Esaminando la gamma di programmi disponibili sia per il C-64 sia per il C-128, si può dire che, con notevoli eccezioni, l'aspetto dello schermo non è sempre curato nel modo più indicato. Inoltre, esiste una comprensibile ma non perdonabile pigrizia dei programmatori per quanto riguarda la funzione HELP. Usando certi eccellenti ma complessi programmi, se l'utente dimentica una delle "regole" del programma, può trovarsi costretto a ricaricare per ritrovare le istruzioni o a sospendere il lavoro per

cercare l'informazione nel manuale. In genere il programmatore avrebbe potuto evitargli questo. Il caso peggiore si trova nei videogiochi importati e tradotti in italiano: spesso vengono semplicemente eliminate tutte le istruzioni tranne i titoli.
Curare un programma da questi punti di vista è facile, specie se si riflette un poco prima di cominciare il lavoro.

3

Input/output

Le novità del BASIC 7.0 rispetto al BASIC 2.0 non consistono in genere in prestazioni che erano del tutto precluse con il BASIC 2.0, ma piuttosto nella possibilità di fare con un solo comando ciò che prima ne richiedeva una lunga serie, oppure nella maggior semplicità della sintassi. Oltre ad un paio di novità assolute, i nuovi comandi per la gestione del dischetto ricadóno in questa seconda categoria: sono più vicini alla normale logica del linguaggio umano. I nuovi comandi funzionano naturalmente anche con il vecchio drive 1541. Il programma DIRTUT esiste in due versioni: sul Lato 1 una versione in BASIC e sul Lato 2 una versione (DISCOTUT) convertita direttamente in p-code dal BASIC senza alcuna modifica. Consente di leggere il direttorio del disco in vari modi. Come si vedrà più avanti, ciò è possibile anche con il nuovo comando DIRECTORY (che è anche più veloce). Il suo scopo è principalmente illustrativo, ma aggiunge alcune possibilità non possedute da DIRECTORY: presenta l'elenco dei file su due colonne sullo schermo tradizionale e su quattro colonne sul monitor a 80 colonne. Ciò è utile per i dischi molto pieni. Un secondo menu consente di visualizzare elenchi selettivi, non solo quelli consentiti con DIRECTORY ma (grazie all'uso di INSTR) anche di file contenenti qualsiasi gruppo di lettere. Il terzo menu consente di eseguire alcuni dei comandi illustrati nel seguito di questo capitolo, spiegandone allo stesso tempo la sintassi (e dando la possibilità di cambiare idea e di non eseguire il comando). L'elenco generale termina con un'indicazione dei blocchi liberi, come DIRECTORY, ma anche di quelli occupati, e avverte se il totale degli uni e degli altri non ammonta a 664 (drive 1541) o a 1328 (1571).

Questo capitolo fornisce informazioni generali relative a:

BSAVE BLOAD DSAVE DLOAD DVERIFY RUN DIRECTORY HEADER APPEND COLLECT CONCAT DCLEAR DCLOSE DOPEN RENAME BOOT.

Come già nel C-16 e il Plus/4, ci sono possibilità notevoli per le operazioni con il dischetto più frequenti, che con il C-64 richiedevano una sintassi abbastanza complessa.

3.1 Drive e unità

L'Appendice L della System Guide fornisce un elenco utilissimo dei nuovi comandi del BASIC 7.0, e un confronto con quelli del C-64 (BASIC 2.0). Sono 27 i comandi-disco del BASIC 7.0, contro gli 8 del BASIC 2.0.
In tutti i comandi che riguardano operazioni nei confronti del disco, possono essere necessari parametri D e U. Il primo (D0 oppure D1) serve solo a chi ha un doppio drive (cioè un solo apparecchio comprendente due drive). Il doppio drive è così raro in Italia che forse nessun lettore di questo libro avrà mai bisogno di usare questo parametro.
Il secondo (U8/U9/...) serve a chi ha più unità (in altre parole, "Unità" per chi sta usando più di un drive singolo, è il numero dell'apparecchio). I default sono ,D0 ,U8: cioè quando il comando è indirizzato alla prima unità (o all'unica che abbiamo). Le parti nuove del BASIC 7.0 infatti seguono in genere questo principio: se in un comando ci sono solo numeri preceduti da virgole per separare i parametri, è necessario inserire almeno le virgole (se non vogliamo cambiare tutti i parametri: per un esempio si veda CIRCLE): se invece un parametro è preceduto da una lettera, si può omettere del tutto (per un esempio si veda PLAY). Ciò vale anche per la B del banco memoria, come in BLOAD e BSAVE.
Questi, insieme a RUN "*programma*" e BOOT (si veda più avanti), sono le novità più notevoli del C-128 per quanto riguarda l'input/output. Verranno presi in considerazione alla fine di questo capitolo. I seguenti comandi nuovi consentono invece di eseguire operazioni che erano già possibili: sono più semplici e facilitano notevolmente l'uso del calcolatore.

3.2 Caricare e salvare su disco

DLOAD "*programma*"

carica un programma da dischetto. Non c'è bisogno di chiudere le virgolette se non segue qualche altra indicazione. Si può specificare da quale drive in un drive doppio (D0 o D1) e da quale apparecchio se si

dispone di più drive singoli (U8, U9...). Usato in modo diretto, carica il programma semplicemente: si userà RUN per eseguirlo. Da programma, invece, carica e mette in esecuzione il nuovo programma. Questo è un "avviamento a caldo" (warm start): i valori delle variabili non vengono perduti. Si veda più avanti RUN "*programma*".
Per esempio:
DLOAD "*programma*",D1,U9
carica *programma* dal drive 1 di un doppio drive collegato come unità 9. Chi invece ha due drive scrive:
DLOAD "*programma*",U9
per caricare *programma* dal secondo drive. Se si omette l'opzione U9, invece, il dischetto da leggere si trova sul drive 8, che è il default.

DSAVE "*programma*"
salva *programma* sull'unità 0, drive 8, se non si aggiungono le opzioni D1, U9, ecc. *programma* deve essere in BASIC: in caso contrario usare BSAVE oppure salvare dal Monitor (S).

DVERIFY "*programma*"
(con opzioni se necessario) verifica *programma* su dischetto.

Questi comandi, è vero, risparmiano anche un po' di fatica. Ma il loro vero pregio è che sono più semplici; è meno facile sbagliare, ed è più facile trovare l'errore, perché sono di più semplice lettura.
Le forme abbreviate di questi comandi sono:

DLOAD dL
DSAVE dS
DVERIFY dV

Si noti bene che, come è indicato anche nel Dizionario di questo volume, l'argomento di questi comandi (argomento indica ciò su cui il comando deve operare) può essere un nome tra virgolette, oppure una variabile stringa. In questo caso la variabile deve essere tra parentesi:

```
P$="PROGRAMMA"
DSAVE (P$)
```

Utilissima è la possibilità di dare al comando RUN un argomento (numero di riga o nome di programma).

RUN fa partire il programma che è già in memoria;
RUN 55 invece, fa partire il programma dalla riga 55;
RUN "*nuovoprog*" è la novità rispetto al BASIC 2.0: ordina al calcolatore di cercare su dischetto il programma *nuovoprog*, di caricarlo e di eseguirlo. Naturalmente si possono dare opzioni come ,D1 oppure ,U8 per indicare drive o unità.

DLOAD e LOAD hanno questa stessa possibilità se usati in un programma. RUN *nuovoprog* può essere usato direttamente da tastiera. Mentre DLOAD e LOAD conservono intatte le variabili, RUN esegue un "avviamento a freddo": cioè azzera la memoria delle variabili.

3.3 Il direttorio - Elenco del file su disco

DIRECTORY

Questo comando lista il direttorio del dischetto, ed è veramente utile grazie agli argomenti che si possono aggiungere. Per esempio:
DIRECTORY "L*
elenca tutti i file (programmi o file, sequenziali, random o relativi) il cui nome inizia con L.
DIRECTORY "*=p
elenca tutti i file di programma... questo è particolarmente utile quando si ha un dischetto molto pieno. Analogamente,
DIRECTORY "*=s
elenca tutti i file sequenziali, e così via.
DIRECTORY "*=p
elenca i programmi i cui nomi cominciano con f.
Esiste, inspiegabilmente, anche il comando CATALOG, che funziona esattamente come DIRECTORY.

3.4 Formattazione del disco

Per formattare un dischetto esiste il comando HEADER. Questo ha la sintassi:

HEADER "*nomedisco*", I*xx*, D*n*, U*n*

dove I*xx* è il numero d'identità del disco, D*n* è il numero del drive, U*n* il numero dell'unità. Tutti e tre possono essere omessi. *xx* sono due caratteri qualsiasi, e *n* è il numero opportuno per il drive e per l'unità. Quindi, per formattare il dischetto che sto usando in questo momento sul drive 9, potrei scrivere:
HEADER "diario",I3B,D0,U9
Il comando richiede certamente un po' di attenzione, ma è molto più comodo del comando necessario con il C-64.
HEADER "*nuovonome*" senza I*xx*
elimina tutti i file dal disco e gli assegna *nuovonome* (che però potrebbe essere lo stesso nome di prima), senza riformattare il disco. È estremamente veloce.

3.5 Altri comandi per il disco

APPEND per aggiungere dati alla fine di un file sequenziale. Apre il file, colloca il puntatore alla fine del file esistente. Si procede quindi con PRINT#. La sua utilità è evidente, ed è facile da usare.

COLLECT per rimettere ordine nel direttorio del disco, eliminare file non chiusi correttamente, ecc. Non deve essere usato su dischetti contenenti file relativi, perché li distruggerebbe totalmente o parzialmente, ma nonostante questa limitazione è di enorme utilità. L'uso periodico di COLLECT fa parte, diciamo, della normale manutenzione del disco quando si sono effettuate varie operazioni di scrittura e di cancellazione: se, leggendo il direttorio con DISCOTUT o DIRTUT, si trova una discrepanza tra il numero dei blocchi liberi/occupati e il totale (664 o 1328) disponibile sul dischetto, è necessario eseguire COLLECT. Prima di farlo, osservare se esistono file impropriamente chiusi (in genere file sequenziali che nel direttorio appaiono con *SEQ o *PRG). In questo caso non si deve usare SCRATCH ma, appunto, COLLECT: se non ci sono file di questo tipo, sarà molto consigliabile eseguire una copia completa del disco (usare il C-128 DOS SHELL, disco fornito con il calcolatore) prima di usare COLLECT. Se per qualche motivo, dopo COLLECT, risultassero mancanti dei dati, sarà forse possibile ricuperarli dalla copia.

CONCAT per concatenare due file (cioè attaccare un file alla fine dell'altro).

DCLEAR chiude i canali tra calcolatore e unità e inizializza il drive. Non è un comando d'uso frequente, in genere una precauzione dopo una serie di operazioni. Se lampeggia la spia d'errore del drive, controllare che tutti i file del lavoro in corso siano stati chiusi. DCLEAR non li chiude, e perciò sarà necessario eseguire invece DCLOSE. Anche PRINT DS$ ferma il lampeggio della spia (e stampa l'eventuale messaggio d'errore: si veda più avanti).

DCLOSE con argomento chiude un file. Senza argomento chiude tutti i file aperti su drive e unità specificati.

DOPEN apre un file su disco. La sintassi di questo comando, che è semplicemente una forma meno complessa di OPEN, non è spiegata chiaramente nella System Guide. Quando si apre un file per la scrittura, la sintassi è

DOPEN #*n*, "*nomefile*",w

in cui *n* è il numero del file e w significa write (scrivere). Come in tutti questi nuovi comandi, se il nome del file è contenuto in una variabile, questa deve essere messa tra parentesi:

DOPEN #*n*, (*f*$)

apre il file numero *n* dal nome *f*$ per la lettura. Ciò che la System

Guide non spiega è come agire con ,w. La forma è
DOPEN #*n*, (*f*$)+",w"
RENAME cambia il nome di un file già esistente su disco.

Il Dizionario alla fine di questo volume fornisce sintassi ed esempi per questi comandi, che per quanto importanti e utili, non richiedono spiegazioni particolari. Il programma DIRTUT offre inoltre la possibilità di implementare i comandi SCRATCH, CONCAT, RENAME, COPY e COLLECT, mentre fornisce un costante aggiornamento sulle condizioni del disco.

3.6 BOOT

Questa parola chiave porta al C-128 un concetto completamente nuovo nella serie Commodore.
Il termine BOOT viene dal termine "bootstrap" (lacci delle scarpe), ed esprime il concetto di "risollevarsi da solo"... ossia, per un calcolatore, di autoprogrammarsi partendo da un minimo assoluto di programma già in memoria. Il C-128 esegue un'operazione BOOT al momento dell'accensione, leggendo direttamente nella propria ROM (Read Only Memory) i dati necessari e trasferendoli, se necessario, in opportune zone della RAM. Anche il PC, o un sistema UNIX, fa la stessa cosa, soltanto che la quantità di ROM è minima (meno di 9 K per il BOOT). Questi calcolatori caricano il resto del sistema operativo dal disco: la piccola ROM serve soprattutto a rendere il calcolatore capace di leggere il disco.
Questo spiega perché il C-128 si avvia più velocemente di un PC: la ROM è più vasta e contiene tutte le informazioni necessarie all'operazione normale del sistema. E leggere la ROM è più veloce che leggere il disco.
Nonostante questo il C-128 è capace di leggere automaticamente dal disco, se questo è già presente nel drive quando si accende il calcolatore. Oppure, se il sistema è già acceso, si può usare il comando BOOT.
Il comando BASIC BOOT provoca in realtà un secondo BOOT (reboot), che può cambiare il sistema operativo escludendo quello della ROM, sostituendolo o ampliandolo, oppure può semplicemente caricare un programma e mandarlo in esecuzione. In altre parole: se si inserisce il disco CP/M e si esegue il BOOT, il C-128 si trasforma praticamente in un altro calcolatore: se invece si inserisce il disco fornito insieme a questo volume, il sistema operativo resta invariato e si carica semplicemente il programma che gestisce la lettura del disco. Entrambi i dischi contenuti nella confezione del C-128 contengono programmi di questo tipo. Uno carica il sistema operativo CP/M, l'altro il DOS SHELL della Commodore. Il primo esclude buona parte del sistema operativo 128: il secondo (a parte il cambiamento della stringa contenuta nel tasto funzione F1) lo supplemen-

ta dando accesso a delle routine (sottoprogrammi) altrimenti inesistenti. Il programma AUTOBOOT MAKER, sul dischetto CBM DOS SHELL, rende facile dotare un disco del caricamento automatico del programma iniziale del disco.

DISCO BOOT PERSONALIZZATO

Una delle applicazioni più semplici e più utili dei comandi RUN e BOOT è la personalizzazione del modo operativo del C-128 al momento dell'accensione. Chi scrive, per esempio, preferisce, per programmare e per effettuare operazioni di housekeeping, uno schermo bianco (grigio chiaro ottenuto con COLOR 0, 16 e COLOR 6, 16), con caratteri neri.
Si può scrivere un programmino che determini i modi di operazione del calcolatore, per esempio:

```
1 print chr$(14) : rem: imposta maiuscolo/minuscolo
2 color 0, 16: color 6,16: color 5,1:
   rem: imposta colori per schermo a 40 e 80 - carattere nero
3 key8, "directory" + chr$(34) + "*=p" + chr$(13):
   rem: direttorio solo programmi
4 key7, "window 0,7,39,24" + chr$(13):
   rem: imposta finestra (40 col) per programmazione.
key8, print"ss" + chr$(13):
   rem: 2 HOME eliminano eventuali finestre
```

Poi si può usare il programma AUTOBOOT MAKER per creare un dischetto BOOT collegato a questo programma.
Il programmino può comprendere anche un menu per caricare altri programmi e mandarli in esecuzione automaticamente, come nei vari menu del dischetto dimostrativo che accompagna questo libro:

```
50 if x=10 then run "prog.2"
```

È meglio usare un dischetto nuovo per questa preparazione. In seguito, si può naturalmente riempirlo di altri programmi.
Le impostazioni create con questo dischetto resteranno in vigore fino a quando non verrà specificata un'impostazione diversa (da tastiera o da programma). Naturalmente varranno solo fino allo spegnimento del calcolatore: alla prossima accensione si dovrà ricaricare di nuovo il profilo utente. Nulla vieta di avere più profili di questo genere. Inoltre, si può organizzare un programma in modo che, quando termina, ricarichi il programma BOOT. Questo può poi lasciare il sistema allo stato iniziale, ed è possibile far sì che l'utente non si accorga nemmeno della reinizializzazione. Il sottoprogramma DEF.ENV, che si carica all'inizio del Menu Mu-

sicale del nostro disco ha lo scopo di riportare ai valori di default i 10 ENVELOPE che definiscono gli strumenti musicali. Così, se un altro programma ridefinisce gli ENVELOPE, non è necessario terminare ogni programma del genere con la ridefinizione. Questa versione si ferma e lista sullo schermo le operazioni che esegue, perché si tratta di un disco dimostrativo, ma ciò avviene soltanto la prima volta che l'operazione è necessaria. Si utilizza una scrittura in due indirizzi di RAM assoluta per evitare la ripetizione. In un'applicazione normale non è necessario che l'utente perda questo tempo: eliminando il LIST e il relativo GETKEY nel programma DEF.ENV, l'utente che entra nel menu musicale non sarà al corrente di ciò che accade. Il programma DEF.ENV ristabilisce i default del sistema, ma potrebbe anche servire a stabilire dei default personali dell'utente. Si veda anche il capitolo PLAY.
Si noti che per caricare il sistema operativo CP/M è obbligatorio il BOOT.

3.7 BSAVE e BLOAD

La System Guide parla di questi comandi soltanto con riferimento agli Sprite (si veda anche il Capitolo 11 di questo libro), e anche se, leggendo tra le righe, si può sospettare l'esistenza di possibilità più ampie, il volume della Commodore lascia il lettore nel dubbio.
In sintesi, le operazioni utili che si possono compiere con i due comandi sono numerose e importanti:

- Leggere il contenuto di una determinata area di memoria e poi creare un file binario (scritto sul disco bit per bit e non byte per byte), riproducente con esattezza tutti i dati.
- Leggere il file e reinserire i dati in questione in memoria, esattamente come prima, oppure in un altro punto.

Il file binario è il tipo di file che viene usato per salvare i programmi. Tra i vantaggi rispetto al file dati salvato byte per byte anziché bit per bit, esiste quello della velocità, specie sul drive 1571. Per il tipo di lavoro che si desidera illustrare nel seguito, l'uso dei comandi è semplicissimo: è però importante capire ciò che stiamo facendo. Prendiamo come esempio un'applicazione che sarà senz'altro tra le più frequenti.
Come è spiegato nei capitoli relativi ai comandi grafici, quando si lavora in modo grafico, 9 K di memoria all'inizio dell'area più normalmente riservata al BASIC, vengono dedicati al bit mapping, cioè alla grafica: il programma (se esiste) viene quindi "rilocato" dopo la fine dell'area grafica.
Gli indirizzi di questo settore di memoria vanno da 7168 a 16383 (da $1C00 a $4000 in codice esadecimale).

10 rem: iniz=7168, fine=16383: banco=0

Supponiamo ora di aver eseguito un grafico complesso che, oltre a lunghi calcoli matematici, richiede anche vari comandi come PAINT e CIRCLE, e che quindi compare lentamente sullo schermo; oppure supponiamo di averlo disegnato in modo diretto e poi di esserci accorti di aver prodotto un capolavoro. In questo secondo caso non esiste affatto un programma e rischiamo quindi di perdere tutto.
Esiste ovviamente la possibilità di non spegnere mai più il calcolatore...

```
10 rem: iniz = 7168, fine = 16383: banco = 0
20 print "SAVE oppure LOAD (s/l)?": getkey a$
30 if a$ = "s" or a$ = "l" then begin
40 :    input" Nome del file"; n$
50 :    if a$ = "l" then 130
60 bend:else 20
110 bsave (n$), b0, p7168 to p16208
115 verify n$,8,1: ?ds$
120 ?"gPremere * per ricaricare il grafico": getkey b$
125 if b$ <> "*" then end
130 graphic 1,1
140 bload (n$), b0, p7168
```

Questo programma non compare sul disco; è una versione di SALVAGRAFICI per salvare grafici in alta risoluzione (GRAPHIC 1/2), e come si vede l'indirizzo finale è diminuito a 16208. Le informazioni così salvate sono sufficienti solo per i grafici ad alta risoluzione. Le maggiori informazioni richieste per il colore in bassa risoluzione (GRAPHIC 3/4) non solo esigono un indirizzo finale di 16383 (che è sufficiente per dare un grafico in cui tutte le zone colorate con la fonte tre hanno lo stesso colore), ma è necessario usare anche una routine simile a quella contenuta nelle righe 5000... di SALVAGRAFICI per salvare, in un altro file (prefisso CF. nel nostro programma), 1 K circa di informazioni dal banco 15. SALVAGRAFICI crea quindi un file (GR.) per i grafici ad alta risoluzione e due (GR. e CF.) per quelli a bassa risoluzione. Un file GR. occupa 37 blocchi (9 K) sul disco, mentre il file CF. ne occupa 5. Il programma riportato qui sopra consente di risparmiare un blocco, registrando un disegno prodotto con GRAPHIC 1/2 in un solo file di 36 blocchi.
Si può caricare SALVAGRAFICI (o questo programmino) anche dopo aver eseguito un grafico con un altro programma o in modo diretto, purché ovviamente non sia stata eseguita un'istruzione GRAPHIC CLR. Il programma riportato qui, dopo aver chiesto se deve leggere un file esistente (1) o salvare uno nuovo (s), chiede il nome del file. Se A$="s", dalla riga 60 va alla 110, dove salva il file con il nome N$. La riga 115 verifica l'operazione (notare che si usa VERIFY "*nomefile*",8,1 e non DVERIFY). Poi

la 120 dà la possibilità di ricaricare il disegno. Se si preme *, la 130 pulisce lo schermo grafico (necessario per essere sicuri che il file sia stato letto e il grafico reinserito in memoria). Poi la 140 carica il file. Non ci sono altre formalità. Se invece si vuole semplicemente leggere un file esistente, la 50 conduce direttamente alla 130.
Poiché il tempo necessario per rivedere un grafico è brevissimo, in molti casi in cui un programma richiede effetti grafici, sarà più efficiente e più comodo registrare i grafici in questa maniera, e poi eliminare dal programma tutte le righe che servivano per crearli. Il programma risulterà più breve e più veloce.
Notare che è necessario specificare inizio e fine (P7168 e P16208 o P16383) nel BSAVE, mentre BLOAD richiede solo il primo valore: infatti carica il file un indirizzo dopo l'altro in ordine progressivo, e la lunghezza del file determina l'indirizzo finale senza altre formalità. BLOAD senza nessun numero dopo il nome del file carica il file a partire dallo stesso indirizzo iniziale dal quale è stato salvato. In realtà, nel programma riportato qui, non è realmente necessario specificare P7168 dopo BLOAD. Il nome del file sarà seguito dalla sigla PRG (il DOS definisce programmi tutti i file binari). Per evitare confusioni può essere una valida idea aggiungere un prefisso speciale al nome del file.

```
45 n$="gr."+n$
```

Aggiungendo questa riga al programma, i file grafici saranno più identificabili nel direttorio, e il comando DIRECTORY "GR.* =P elencherà sullo schermo soltanto i file con questo prefisso. SALVAGRAFICI usa questa formula. Sul disco si utilizza invece il prefisso "SP." per i file di sprite. Si vedano anche i capitoli relativi agli sprite, e il programma DIRTUT/DISCOTUT.
I numeri (iniz=7168, fine=16383, banco=0) devono ovviamente essere corretti. Se il primo è inferiore a 7168, si possono avere problemi di vario tipo (un'inspiegabile ridefinizione dei tasti funzione, per esempio): un numero finale superiore a 16383, invece, invaderà l'area del BASIC, rovinando il programma in memoria. Per una spiegazione più completa del concetto di bit map e dello schermo grafico in genere si vedano il Capitolo 7 e il programmino ORA ESATTA (Menu Comandi Grafici) che ricarica in 6 secondi un disegno che ELLISSE (sullo stesso menu) impiega oltre un minuto a disegnare. Il prezzo che si paga è l'occupazione di ben 36 blocchi (9 K) sul dischetto.
Prima di utilizzare BSAVE è necessario spegnere gli sprite eventualmente accesi (FOR T=1 TO 8: SPRITE T,0: NEXT). In caso contrario si possono avere svariati e sgraditi effetti sul disco, e quando si ricaricano i file vengono messi fuori uso i tasti funzione e HELP. Questo è una lacuna nel sistema operativo: meno accettabile è la lacuna nella System Guide, che non suggerisce la soluzione.

Prima di utilizzare BLOAD per caricare un disegno è necessario eseguire GRAPHIC 1: GRAPHIC 0: questo serve a spostare il programma in un'area di memoria che non verrà occupata dal grafico. In caso contrario il programma sarà rovinato e si può avere il blocco del sistema.

3.8 Stampanti

Viene spontaneo a questo punto domandarsi perché, con tutte le nuove possibilità offerte per il drive, non ci sia niente di particolare per la stampante. Per esempio, per facilitare la stampa di brevi pezzi di testo senza dover caricare un apposito programma, quando l'eleganza non importa molto ma si vuole almeno che il testo venga scritto sulla carta e non sul rullo, sarebbe molto utile disporre di un comando come LPRINT del BASIC Microsoft, preferibilmente migliorato, con una sintassi del tipo:

LPRINT *lp*, *nc*, "...

con *lp*=numero di righe per pagina, *nc*=numero di caratteri per riga, e con default 55 e 80 rispettivamente. Le istruzioni necessarie per formattare la pagina, quando si vuole stampare qualche cosa ad hoc, sono abbastanza complesse e si può essere portati a evitare l'operazione, con il rischio di non vedere un errore, oppure a fare una stampa "brutale" e scomoda che non rispetta il formato pagina. Inoltre, non sarebbe stato impossibile fornire anche il C-128 di un tasto PRINT SCREEN, come quello del PC: è sufficiente premere questo tasto per inviare alla stampante l'intero contenuto dello schermo.
Il lettore potrebbe domandarsi perché questo volume non riservi un capitolo all'uso delle stampanti: la risposta è che (a parte CMD e OPEN) il BASIC 7.0, come i suoi predecessori, non prende in particolare considerazione la stampa. Inoltre, le stampanti, sia di marca Commodore che di altra marca, presentano un'infinità di differenze. Si rischierebbe quindi di creare soltanto confusione. Un consiglio importante a chi acquista qualsiasi stampante è di leggere il manuale d'uso prima dell'acquisto e, se si richiede dall'apparecchio prestazioni grafiche, preferire una stampante corredata dell'opportuno software (screen dump). Questo software deve inoltre essere specifico per il C-128 e non per C-128 in modo C64.
Se insieme alla stampante si acquista anche un word processor o database, controllate che la stampante sia supportata da questo software. Non conviene fidarsi del rivenditore: controllate di persona leggendo il manuale e, se non siete più che sicuri, non acquistate.

Le maggiori novità del BASIC 7.0

Questo capitolo discute varie nuove voci del BASIC che meritano di essere messe in rilievo anche se sono trattate nel Dizionario alla fine del volume. Sono:

GETKEY - IF... THEN... ELSE - DO WHILE/UNTIL...LOOP/EXIT
BEGIN...BEND - SLEEP - AUTO - DELETE - RENUMBER - HELP
TRON/TROFF - TRAP - ERR$ - ER - EL - DS$ - DS

4.1 GETKEY

Nelle versioni precedenti (escluso il BASIC del C-16 e del Plus/4), per far sì che il calcolatore attendesse la pressione di un tasto (cioè per fermare un processo), bisognava creare un loop, così:

```
100 get a$: if a$=""then100
```

Questo ferma il programma alla riga 100 finché l'utente non preme un tasto (senza l'IF il GET non aspetta). GETKEY elimina la necessità dell'espressione IF A$="". Ma non è tutto: allo statement condizionale IF... è stato aggiunto il potentissimo ELSE ("altrimenti"):

```
100 getkey a$
110 if a$="s" then 120: else end
120 print "hai risposto di si"
```

Questo programmino insiste sulla risposta "si". Se si risponde di no il programma finisce.
GETKEY è molto semplice da usare, e di utilità frequente. La System Guide dà altri esempi. GETKEY può anche chiedere un carattere numerico:

```
100 getkey a$, b, c, d, d$
```

attende quindi un carattere alfanumerico, tre numerici e un altro alfanumerico.

4.2 ELSE

Le potenzialità di ELSE vanno molto più in là dell'esempio che ho dato poco sopra.

```
110 if a$="S" then 120: else if a$="n" then 130: else if a$="!"
then gosub 33000
```

La System Guide dà ovviamente altri esempi: è facile imparare ELSE. Semplicemente: se l'espressione che segue l'istruzione IF è falsa, il programma esegue quella successiva all'ELSE, altrimenti la ignora e procede alla successiva riga di programma.

4.3 DO... LOOP

Loop, come termine generale, significa circolo chiuso (generalmente condizionato): il calcolatore va avanti a fare una cosa finché ...
È senz'altro familiare la costruzione:

```
FOR T=1 TO 10: ? "CIAO": NEXT T
```

Questo è un tipo di loop indispensabile, senza il quale il calcolatore servirebbe a poco: risparmia righe e righe di programma. Ma ha i suoi difetti. Finché esiste un numero preciso di ripetizioni, e lo possiamo specificare, siamo a cavallo... ma se non sappiamo quante volte possa essere necessario fare una cosa, come la mettiamo? Possiamo arrangiarci in molte maniere, con una serie di IF, decrementando T (pericoloso), eccetera. Ma servono molte righe di programma, si consuma molta memoria (anche se questo non è in genere un problema sul C-128) e il programma è più lento e molto meno facile da capire quando lo leggiamo.

Qui vengono in nostro aiuto le nuove specifiche:

DO UNTIL WHILE LOOP

```
110 x = 1
120 do while x < 6
130 :  play "o5c tl o4 g e c"
135 :  get a$
140 :   if a$ = "+" then x = x + 1
145 :   if a$ = "-" then x = x - 1
146 :  char,30,5," x = " + str$(x),1
150 loop
```

Questo programmino fa quanto segue:

110 stabilisce il valore iniziale di x
120 fa quanto segue mentre x è inferiore a 6
130 suona le note Do Sol Mi Do (arpeggio discendente)
135 controlla se è stato premuto un tasto
140 se il tasto è "+" allora x aumenta di 1
145 se il tasto è "−", diminuisce di 1
146 stampa il valore di x sullo schermo
150 torna a DO.

Questo si potrebbe anche esprimere un po' diversamente, con quasi lo stesso risultato:

```
110 x = 1
120 do
130 :  play "o5c tl o4 g e c"
135 :  get a$
140 :   if a$ = "+" then x = x + 1
145 :   if a$ = "-" then x = x - 1
146 :  char,30,5," x = " + str$(x),1
150 loop while x < 6
```

L'effetto è lo stesso, anche con il WHILE dopo il LOOP, con una differenza che può essere importante: nel primo esempio, se x fosse già pari o superiore a 6, il loop non verrebbe eseguito nemmeno una volta: dopo aver letto la 120, il programma salterebbe alla prima riga dopo la 150. In questo secondo caso, invece, eseguirebbe le righe 130-150 una volta.
Analogo e altrettanto facile è il comando UNTIL.

```
210 x = 10
220 do until x < 5
230 :  play "o4 t2 c g  e o5 c"
```

```
235 :  geta$
240 :   if a$ = "+" then x = x + 1
245 :   if a$ = "-" then x = x - 1
246 :  char,30,14," x = " + str$(x) + " ",1
250 loop
```

I due comandi sono il diritto e il rovescio di una stessa medaglia. Mentre il primo dice di fare il lavoro MENTRE x<6, il secondo dice di farlo FINO A QUANDO sarà inferiore a 5. Quindi per uscire dal primo loop si dovrà premere "+", mentre qui dovremo premere "−". Non c'è nessuna differenza funzionale, ma a seconda di quel che vogliamo fare, potremo risparmiare spazio con l'uno o l'altro.
Esiste anche il comando EXIT che si usa così:

```
60 do
65 :   play"v1 o4 t9 c e g o5 c"
70 :   get a$
75 :   if a$ = "*" then exit
80 loop
```

Questo loop continua a suonare le note se non premiamo "*".
Potremmo usare anche DO UNTIL A$="*" oppure LOOP WHILE A$<>"*", ecc., e la scelta dipenderà dalle circostanze: EXIT può essere vantaggioso per casi eccezionali.
Possiamo avere molti loop anche di tipo diverso annidati (nested) l'uno dentro l'altro; dare un rientro diverso alle specifiche di ciascun loop rende il tutto molto più facile da capire. Un esempio schematico è presentato all'inizio del Capitolo 2, e i programmi sul disco ne presentano innumerevoli esempi.

4.4 BEGIN e BEND

Un'altra istruzione nuova. Anche in questo caso, non si tratta di una possibilità completamente nuova: piuttosto ci risparmia della fatica, e rende molto più chiara la struttura dei programmi. Per capire in quale modo, bisogna prima di tutto considerare le routine usate con GOSUB.
Lo scopo di GOSUB è di rendere disponibile a una qualsiasi linea di programma una stessa routine. Come suggerisce la parola routine, si dovrebbe trattare di procedure ben precise (per esempio dividere x per y o stampare una riga sulla stampante o disegnare un cerchio), che sarebbe assurdo scrivere mille volte.
Nonostante questo, nelle versioni precedenti del BASIC CBM, si è usato GOSUB anche per operazioni da eseguire una sola volta, perché l'istru-

zione condizionale IF valeva per una sola riga. Cioè, se la condizione indicata nell'IF era falsa, il programma continuava alla riga successiva. Una soluzione era di questo tipo:

```
100 if a$="si" then ...
110 if a$="si" then ... ecc.
```

Qui, quando il calcolatore arriva a 100, se a$ non è uguale a "si", prosegue senza fare niente finché non trova una riga che non richiede che a$="si".
Più elegante è

```
100 if a$<>"si" then 200
```

Queste possibilità non sono sempre così facili da capire, naturalmente, e non sempre la scelta è fra un "si" e un "no", ma può essere molteplice. Da qui nasceva l'abitudine di usare un GOSUB: infatti si poteva ridurre il numero di istruzioni IF..., e comunque tenerle vicine l'una all'altra; comunque il sistema rendeva più complessa la lettura del programma (e i GOSUB consumano anche memoria).
GOSUB è un comando potentissimo, ma il suo compito vero è di dare accesso a sottoprogrammi veri, cioè da usare molte volte nel corso di un programma, non per costringerci ad andare a leggere da qualche altra parte del programma, ogni volta che un IF contiene più di 160 byte di comandi.
Ecco quindi un'istruzione che permette di rendere molto più chiara la struttura del programma, senza artifici particolari.
La forma dell'istruzione BEGIN...BEND è in genere come segue. Cominciando con un IF:

```
begin2.list
100 print"Inserire una breve stringa (nome, ecc.)":input x$
200 print"C = CENTRARE D = destra: S = sinistra"
205 getkey a$
210 if a$ = "c" then begin
220 :  x=len(x$)
230 :  do until len(x$) = int(20+(x/2))
240 :     x$= " " + x$
250 :  loop
260 :  goto340
265 bend
270 if a$ = "d" then begin
280 :  x=len(x$)
290 :  do until len(x$) => 39
300 :     x$= " " + x$
310 :  loop: goto 340
```

```
320 bend
330 if a$ <> "s" then 205
340 print x$
```

Si tratta di centrare, o di allineare a destra o a sinistra, la stringa x$: l'utente sceglie C, D o S. A seconda della scelta il programma esegue uno dei due loop (un loop per S non è necessario, perché x$ verrà comunque stampato a sinistra se non vengono inseriti degli spazi all'inizio). Qui l'alternativa è di usare due GOSUB, che avranno lo stesso effetto ma ci costringeranno ad andare a cercarli in qualche altra sezione del programma. E anche supponendo che riuscissimo a scrivere le righe da 210 a 265 e da 270 a 320 in 2 righe, per tenerle nella posizione "naturale", BEGIN... BEND consente di renderle molto più leggibili.
Per questi nuovi loop vale una raccomandazione. Controllare sempre che il programma esca effettivamente dai loop in maniera regolare. In particolare, se il loop viene interrotto da TRAP è necessario tener presente che il calcolatore registra i dati d'inizio e di fine dei loop attivi: se quindi non terminano correttamente si ha una situazione in cui il programma esegue un loop dentro un loop dentro un loop... fino a riempire tutta la memoria. Si ha allora un blocco del programma. Significa che bisogna organizzare i TRAP in modo che il loop venga chiuso dopo l'errore, oppure liberare la memoria con CLR.

4.5 Il rallentatore

Una breve nota su un comando semplicissimo. I programmi veloci sono senz'altro utili, ma spesso e volentieri può capitare che ci serva invece rallentarli o fermarli, per esempio per presentare pagine di testo una alla volta, o per evitare che l'utente batta troppi tasti troppo in fretta e perda il filo logico delle operazioni.
Sugli altri calcolatori Commodore dovevamo fare un loop del tipo:

```
100 for t=1 to 600: next t
```

Questo dà un ritardo di 10 secondi (600 sessantesimi di secondo): il C-128 ha un comando molto più utile:

```
100 sleep 10
```

Significa semplicemente "dormi per 10 secondi"... fino a 65535 secondi se si vuole. Non indispensabile, ma ottimo quando desideriamo programmi semplici da leggere.

4.6 Aiuti alla programmazione

Avendo parlato di aiuti alla programmazione, concludiamo con un paio di osservazioni su quanto ci offre il C-128. Come si è già detto, si poteva sperare in qualcosa di più, ma non sono trascurabili i 7 comandi, nuovi rispetto al C-64 ma non rispetto al C-16 e al Plus/4.

AUTO

Con questo comando (usato solo in modo diretto, ovviamente) si inserisce automaticamente il numero di riga successivo quando si preme RETURN alla fine della riga precedente. Per esempio:

```
AUTO 10
```

numererà la riga di programma successiva aumentando di 10 il numero precedente. Una confusione può nascere dal fatto che non ci presenta il primo numero: in altre parole, dopo aver battuto quel comando, si deve battere la prima linea, numero compreso. Allora, si avrà una situazione del genere:

```
AUTO 10
5 ?"IL PIU' GRANDE PROGRAMMA DEL MONDO"
15
```

Il cursore attende dopo il numero 15. Si può inserire qualsiasi intervallo che si desidera, anche AUTO 1, e partire da qualsiasi numero. AUTO senza argomento termina la numerazione automatica.

DELETE

Significa cancellare: sul C-64 l'unico modo era di battere il numero di ognuna delle righe che si volevano eliminare.

DELETE 45	cancella la riga 45
DELETE 45-99	cancella da 45 a 99
DELETE -45	cancella fino a 45
DELETE 45-	cancella da 45 in avanti

In tutti i casi sono compresi il primo e l'ultimo numero. In nessuno di questi esempi sopravvive la riga 45 (né la 99).

RENUMBER

Significa rinumerare. La sintassi di questo comando è più complessa. E in effetti è resa anche più complessa da un errore di stampa nella System Guide.
È opportuno salvare il programma subito prima di rinumerarlo (può essere una buona idea non solo per sicurezza, ma anche nel caso che la nuova numerazione ci faccia perdere un momento l'orientamento e non riusciamo più a trovare la riga sulla quale stavamo lavorando).
La sintassi è come segue:

RENUMBER *nuovo numero iniziale, incremento,*
vecchio numero iniziale

Le espressioni:

RENUMBER
RENUMBER 10
RENUMBER 10, 10

danno tutte una nuova numerazione a partire da 10 con incrementi di 10 (perché il default per l'incremento è appunto 10). Invece:

RENUMBER 100, 5

rinumera a partire da 100 con incrementi di 5.
Ora supponiamo di avere un programma che vogliamo rinumerare solo a partire da un certo numero.
RENUMBER 2000, 10, 1000 lascia in pace tutti i numeri prima del 1000, e rinumera a partire dal 2000, e con incrementi di 10, tutti quelli successivi. Non c'è da temere per i GOSUB e per altri rimandi o salti di linea. Il calcolatore li tiene in mente e li aggiorna. Attenzione: se noi abbiamo sbagliato e abbiamo inserito un riferimento a una riga inesistente (per errore o perché non l'abbiamo ancora scritta), il programma NON viene rinumerato e riceviamo il messaggio:

UNRESOLVED REFERENCE ERROR

Le virgole in RENUMBER sono indispensabili, ma non i numeri tra l'una e l'altra. Quindi:

RENUMBER ,3

rinumera con intervalli di 3 a partire dal default 10.

RENUMBER 100, ,223

rinumera a partire da 100 le vecchie righe 223 e successive, con l'intervallo di default 10.
Se si rinumera solo la parte finale del programma, sarebbe un errore scrivere, per esempio:

RENUMBER 10, 10, 1000

se esistessero righe con numeri inferiori a 1000. Per proteggere il programma, il sistema dà un messaggio d'errore e non ubbidisce.
RENUMBER da solo rinumera dal 10 con incrementi di 10... e così via.
Un difetto di RENUMBER è che non riconosce LIST in un programma: molti programmi del nostro dischetto listano alcune righe del programma stesso. Non si è potuto rinumerare questi programmi con intervalli ordinati, perché RENUMBER lascia i numeri vecchi.

HELP

Di questo abbiamo già parlato nella sezione dedicata alla tastiera; si veda anche il programma KEY. HELP, quando vi è un errore, stampa sullo schermo la riga in cui si è verificato, evidenziando in negativo (o con sottolineatura sullo schermo ad 80 colonne) il resto della riga a partire dal punto in cui il programma è andato in tilt. La sottolineatura, sullo schermo a 80 colonne, non consente di distinguere tra il punto e la virgola. Se non si riesce a capire quale sia l'errore, sarà opportuno listare la riga in modo normale per vedere se non c'è un punto al posto di una virgola (o punto e virgola al posto di un due punti).
Se invece si stava lavorando con il floppy in quel punto, la cosa da fare è battere:

PRINT DS$

per sapere il tipo di errore che si è verificato sul drive. HELP non funziona per gli errori nei comandi battuti in modo diretto: possiamo invece usare ERR$ (si veda più avanti).
Importante: non correggete la riga in negativo prodotta da HELP; usate LIST per avere una riga normale e correggere questa. Altrimenti in certi casi la riga corretta risulterà piena di caratteri in negativo, e questo può bloccare il programma.

TRON/TROFF

Significa *trace* (TRaceON/TRaceOFF) e, come suggerisce il nome, aiuta a rintracciare. Si usa in modo diretto o nel programma stesso. Prima di eseguire un'istruzione, TRACE presenta tra parentesi il numero della riga (quindi se una riga contiene più istruzioni separate da un due punti, questo numero appare più volte). In questo modo si ha un'idea della posizione nella quale il programma entra in crisi. Se si ha già un sospetto si può inserire TRON/TROFF all'inizio e alla fine della zona sospetta. Per esempio, supponiamo di sospettare l'esistenza di un "baco" in una subroutine:

```
100 tron: gosub360: troff
```

ci farà vedere il momento in cui l'errore accade, senza riempire lo schermo di numeri per tutta la durata del programma. Questo fenomeno, tra l'altro indispensabile dato lo scopo dell'operazione, rende TRACE non idoneo a certi rintracciamenti, in particolare nella ricerca di errori durante la formattazione di stampa su schermo. In questi casi si può agire in modo spiccio e inserire uno STOP subito dopo il punto in cui sospettiamo che si verifichi l'errore.

TRAP

Intrappola o intercetta gli errori prima che il programma si fermi, sia per un errore di programmazione, sia quando i dati forniti dall'utente potrebbero mandarlo in tilt. Un valore impossibile (divisione per zero, per esempio) può fermare il programma. TRAP evita questo.

```
90 trap 1000
100 input "Indica il divisore";a
110 1=n/a
120 ................: end
1000 print "valore inaccettabile": resume 100
```

Anche lo STOP fa scattare TRAP: questo può essere molto utile; si vedano tutti i programmi compresi nel dischetto di questo volume, in particolare la serie DRAW, in cui la pressione di STOP costituisce l'unico modo per uscire dal loop principale che utilizza il joystick. Nei programmi compilati STOP non ferma il programma ma dopo GET o GETKEY si può usare IF A$ ="*c*" THEN... per superare l'ostacolo ("*c*" è il simbolo prodotto da CONTROL-STOP in modo virgolette). RESUME è il complemento ovvio di TRAP: significa "riprendere". Senza argomento (nell'esempio l'argomento

è il numero 100) riproverebbe la linea 110 (non saggio se a=0, poiché si ripeterebbe all'infinito). Un numero fa riprendere dalla riga corrispondente, RESUME NEXT porta invece al comando successivo. La linea 1000 potrebbe essere anche condizionale:

```
1000 if er=10 then ......
1010 if ds=51 then ......
1020 if el=200 then ......
```

ERR$ e ER

ERR$(*n*) è una matrice permanentemente in memoria che contiene le stringhe corrispondenti a tutti i messaggi d'errore. La routine:

```
FOR I=1 TO 41: PRINT I; ERR$(I): NEXT
```

stamperà tutti i messaggi d'errore. Sostituendo ER a I otterremo il messaggio relativo all'errore più recente; l'espressione è quindi PRINT ERR$(ER); potremmo usarlo alla riga 1000:

```
1000 print err$(er): sleep 10: run
```

Questo farebbe vedere il messaggio sullo schermo per 10 secondi, poi farebbe ripartire il programma dall'inizio. Ciò può essere la soluzione migliore se il programma non è lungo, oppure se è appena cominciato. Potremmo quindi usare EL (error-line) che dà il numero della riga in cui si è verificato l'errore:

```
1000 if el<100 then run: else .....
```

In questo caso, se il programma è appena all'inizio, riparte: altrimenti ... fa un'altra cosa.
TRAP è una versione più sofisticata dell'istruzione ON ERROR GOTO, usata in varie versioni del BASIC. Può essere usata una volta all'inizio del programma, e valere fino alla fine, mentre ovviamente può essere rispecificata più volte nel corso del programma stesso. Per esempio un GOSUB può ridefinire una nuova trappola nella prima riga della subroutine e rispecificare quella originale nell'ultima, prima del RETURN.
Un'imperfezione è che TRAP non intercetta un errore FILE NOT FOUND quando, da programma, si dà il comando RUN "*nomefile*": il programma si ferma.
Il più delle volte, TRAP, usato sapientemente, mette il programma al riparo dagli imbecilli. Ed è altrettanto efficace, e qualche volta più com-

prensibile, usare una trappola invece di TRON/TROFF mentre si sta collaudando il programma, evitando quasi sempre il blocco del sistema.

DS e DS$

Disk Status: i mezzi corrispondenti ad ER ed ERR$ per conoscere lo stato del DOS.
Sono rispettivamente il numero dell'errore (se c'è errore, perché DS=0 significa OK), e la stringa che dà la spiegazione dell'errore ("72 DISK FULL", per esempio: si veda l'Appendice B della System Guide). Non c'è in questo caso la stessa sintassi che si usa con ERR$.

```
PRINT DS$
```

è sufficiente. La Commodore fa le cose in modo diverso per tenerci svegli? ERR$ ed ER rappresentano comunque un notevole miglioramento rispetto alla funzione ST (STatus).
Quando si verifica una condizione di errore sul disco, lampeggia la spia del drive, ma se la causa dell'errore non esiste più, l'operazione successiva dovrebbe svolgersi correttamente. PRINT DS oppure PRINT DS$ ferma il lampeggio.

ST (STATUS)

Quest'altra funzione riporta anch'essa le condizioni di errore relative all'input/output. Come si è già osservato, è superato da altre funzioni più facili da usare. Un uso importante rimane, però: se si desidera leggere i caratteri contenuti in un file e non si conosce la struttura di questo, ST consente di leggerlo con GET#.

```
10 dopen#1,"file
20 do until st
30 :   get#1,a$
40 :   print a$;
50 loop
60 dclose
```

Finché ST=0 questa routine continua a leggere un carattere per volta e a stamparlo sullo schermo. Quando si raggiunge la fine del file, ST assume il valore 64 e il loop termina. Questo principio è usato dai programmi TYPE/MORE e FIND (Lato 1 del disco).
Infine osserviamo che alcuni di questi comandi possono non funzionare

in un programma compilato: se richiedono un numero di riga, è necessario tener presente che il compilatore elimina il numero di riga. Esistono opzioni in certi compilatori per mantenere una tabella di corrispondenze, ma producono necessariamente un programma più lungo. È quindi opportuno eliminare espressioni del tipo IF EL<100 in un programma da compilare: TRAP 1000 dovrebbe invece funzionare in ogni caso, e per il resto è in genere possibile ottenere il risultato voluto con espressioni alternative. Consultare il manuale del compilatore.

5

PRINT USING, PUDEF, CHAR

Supponiamo di voler creare un programma per comporre tabelle, sullo schermo o con la stampante. Verrà definito un determinato numero di colonne di dati, che potranno essere più o meno larghe a seconda della larghezza dello schermo (80/40) o della carta: i dati da inserire in ciascuna colonna possono essere alfabetici o numerici, e potremo aver bisogno di allineare i dati alfabetici a destra o a sinistra, o centrarli, come anche di definire il numero delle cifre da stampare a destra della virgola nei dati numerici. Potremmo voler stampare i segni $, + o − prima dei numeri, e così via. Potremmo anche voler stampare il segno £ al posto del $, e una vera virgola al posto del punto anglosassone. PRINT USING, insieme a PUDEF, riesce a rispondere a tutte queste esigenze in maniera eccellente. Si tratta di due comandi eccezionalmente potenti.
Sul Lato 1 del dischetto, il programma DIM (che stampa una tabella contenente una matrice di stringhe) trarrebbe vantaggi notevoli dall'uso di PRINT USING: l'unico motivo per cui non è stato usato è quello di non distrarre il lettore dal vero scopo di questo programma.

5.1 Generalità su PRINT USING e PUDEF

La sintassi è:

PRINT[# *n*,] USING "*formato*";

Il parametro #*n* è il numero del file. Se presente, manda l'output su un dispositivo diverso dal video (disco, stampante, modem): se è assente, il risul-

tato appare sullo schermo. La stringa *formato* definisce il modo in cui devono essere stampate le variabili (stringhe o numeriche) che seguono.
Il *formato* è composto da simboli speciali che definiscono il modo in cui dovranno essere trattati i caratteri nella posizione corrispondente in un'altra stringa o in una variabile numerica. Può essere posto subito dopo la parola USING, oppure può essere definito separatamente come variabile stringa (per esempio, a$="#=##########"), e poi inserito allo stesso posto nell'istruzione. Questo significa che si può costruire un repertorio di formati da usare di volta in volta, che possono essere contenuti anche in una serie di DATA.
PUDEF consente di prendere certi caratteri contenuti nella variabile stringa o numerica che si deve stampare secondo il formato definito, e di cambiarli in certi altri. Ma è opportuno vedere prima che cosa possiamo fare con PRINT USING.
I simboli disponibili sono: # + − . ,$ ^^^^ = >
Tra questi, # + − . ,$ e ^^^^ possono essere usati con variabili numeriche, mentre i tre simboli # = > vanno con le stringhe. Il segno # riserva una posizione per un carattere: il numero dei # deve essere almeno pari al numero di caratteri desiderato, altrimenti non verrà stampato nessun numero ma solo una serie di * * * *, mentre una stringa verrà troncata.

```
pr.using.l.list
10 rem: PRINT using - primo test -
20 rem: allineare valori numerici sul
30 rem: punto decimale
40 rem: prima definisco tre valori
60 x = 1.2345
70 y = 12345
80 z = 12345.6789
90 rem: ora proviamo un formato
100 PRINT using "#####.#####"; x
110 PRINT using "#####.#####"; y
120 PRINT using "#####.#####"; z
150 rem: ora arrotondiamo i decimali
200 PRINT using "#####.#"; x
210 PRINT using "#####.#"; y
220 PRINT using "#####.#"; z
250 rem: aggiungiamo un + e uno spazio precedenti
300 PRINT using " +#####.##"; x
310 PRINT using " +#####.##"; y
320 PRINT using " +#####.##"; z
350 rem: adesso mettiamo i numeri come dollari senza cents
400 PRINT using " $######"; x
410 PRINT using " $######"; y
420 PRINT using " $######"; z
450 rem: con cents e con il $ attaccato al valore
```

```
500 PRINT using " #$######.##"; x
510 PRINT using " #$######.##"; y
520 PRINT using " #$######.##"; z
```

Questo programma stampa:

```
    1.23450   [tutte le cifre con
12345.00000   [l'aggiunta di uno 0
12345.67890   [finale

    1.2       [una sola posizione decimale
12345.0       [il primo e' arrontondato in basso
12345.7       [il terzo e' arrotondato in alto

 +    1.23    [il + aggiunto
 +12345.00    [in posizione fissa: si poteva
 +12345.68    [usare qualsiasi carattere

 $     1      [$ aggiunto in posizione
 $ 12345      [fissa: osservare gli
 $ 12346      [arrotondamenti

      $1.23   [arrotondati a 2 posi-
  $12345.00   [zioni: $ non piu' in po-
  $12345.68   [sizione fissa
```

I commenti in parentesi quadre sono stati aggiunti dopo (non sono prodotti dal programma).
Più importante, questi listati dei risultati dei vari programmi sono pure stati prodotti da PRINT USING nella forma disponibile per stampante e disco, PRINT# *n*, USING. Si è semplicemente trattato di modificare provvisoriamente il programma nel modo seguente:

```
100 PRINT #1, USING "#####.#####;"
```

Lo scopo di questo era duplice: verificare se si riusciva a scrivere un file su disco utilizzando questo metodo di formattazione e riprodurre in questo libro il preciso risultato che era apparso sullo schermo, senza dover fare una penosa ricostruzione. Notare che è indispensabile la virgola dopo il numero del file.
Ecco ora, senza tutte le REM, un secondo programma che usa gli stessi formati per ottenere esattamente gli stessi risultati di prima, definendo i formati come stringhe al di fuori dell'istruzione PRINT USING. Non sono riprodotti i risultati, visto che sono identici.
Il menu dei comandi vari del dischetto allegato a questo libro contiene i programmi riprodotti in questo capitolo nella versione che dà i risultati

sullo schermo (a 40 o a 80 colonne). Sono stati collegati insieme per formare un unico programma; risultano quindi diversi i numeri delle righe.

```
pr.using.2.list
10 rem: PRINT using - secondo test
20 a$ = "#####.#####"
30 b$ = "#####.#"
40 c$ = " +#####.##"
45 e$ = " #$######.##"
50 d$ = " $######"
60 x = 1.2345
70 y = 12345
80 z = 12345.6789
100 PRINT using a$; x
110 PRINT using a$; y
120 PRINT using a$; z
200 PRINT using b$; x
210 PRINT using b$; y
220 PRINT using b$; z
300 PRINT using c$; x
310 PRINT using c$; y
320 PRINT using c$; z
400 PRINT using d$; x
410 PRINT using d$; y
420 PRINT using d$; z
500 PRINT using e$; x
510 PRINT using e$; y
520 PRINT using e$; z
```

Il prossimo esperimento comprende la stampa di più variabili sulla stessa riga, usando uno o più spazi all'inizio della stringa di formattazione per produrre uno spazio tra le colonne.
In una tabella che contiene anche colonne di parole (che è frequentissimo), possiamo posizionare le stringhe in uno di questi tre modi:

bandiera sinistra — è il default
centrate — si usa il segno =
bandiera destra — si usa >

Questi due segni servono solo per le stringhe: i numeri vanno sempre allineati a destra, naturalmente. E se per caso si vuole centrare un numero, nessuno ci impedisce di farlo tramite una espressione del tipo X$= STR$(X), e poi centrare la stringa X$.
Ora cerchiamo di fare un altro passo avanti e di completare l'esplorazione, almeno a questo livello, dell'istruzione PRINT USING.

```
pr.using.3.list
10 rem: PRINT using - terzo test -
60 x$ = "Milano": x = 12345.156
```

```
70 y $= "New York": y = 98765
80 z$ = "Megagalattiche": z = 9.12345
90 rem: stampiamo le 3 stringhe allineate a destra
100 PRINT using ">##################"; x$
110 PRINT using ">##################"; y$
120 PRINT using ">##################"; z$
150 rem: stampiamo anche i numeri e centriamo le stringhe
200 PRINT using "=###############.##"; x$, x
210 PRINT using "=###############.##"; y$, x
220 PRINT using "=###############.##"; z$, z
250 rem: aggiungiamo un + e uno spazio precedenti.
        Allineamento a sinistra per le stringhe
300 PRINT using " +#####.##"; x$, x
310 PRINT using " +#####.##"; y$, y
320 PRINT using " +#####.##"; z$, z
330 rem: la 320 tronca Megagalattiche
350 rem: mettiamo i numeri come dollari senza cents
400 PRINT using "#$###############"; x$, x
410 PRINT using "#$###############"; y$, y
420 PRINT using "#$###############"; z$, z
450 rem: mettiamo la stringa dopo
500 PRINT using "#$##############.#"; x, x$
510 PRINT using "#$##############.#"; y, y$
520 PRINT using "#$##############.#"; z, z$
535 rem: staccare la stringa dalla cifra
600 PRINT using " #$##############.#"; x, x$
610 PRINT using " #$##############.#"; y, y$
620 PRINT using " #$##############.#"; z, z$
```

```
            Milano
           New York
     Megagalattiche

       Milano              12345.16
      New York             12345.16
    Megagalattiche             9.12

 Milano    +12345.16
 New York  +98765.00
 Megagalat +    9.12

Milano                $12345
New York              $98765
Megagalattiche            $9

          $12345.2Milano
          $98765.0New York
              $9.1Megagalattiche

           $12345.2 Milano
           $98765.0 New York
               $9.1 Megagalattiche
```

Le righe 400-420, in un primo tentativo, non avevano il # prima del $: in questo caso si otteneva sia "$Milano" sia (più logicamente) "$12345". Per motivi di spazio, è stato eliminato questo esempio.
La stringa "Megagalattiche" viene troncata perché non ci sono abbastanza # a disposizione. Anche centrando con = o allineando a destra con >, si ottengono le stesse lettere (e non, come si poteva temere, "galattiche)".
Come per il $, se si mette un # prima del segno +, quest'ultimo si attacca al numero invece di restare in una colonna fissa.
Ora esaminiamo il campo con ^^^^. In realtà deve sempre essere #^^^^. Questo stampa un valore numerico in notazione scientifica, come per esempio:

5.543543 e +3

Devono sempre essere 4^ preceduti da un #. Il simbolo ^ si ottiene con il tasto della freccia verso l'alto (accanto a RESTORE sulla tastiera).

```
pr.using.4.list
10 rem:test print using con ^^^^
60 x = 1/3
70 y = 1234
80 z = 9999*999999
90 a = "+##########^^^^ "
100 print using a$; x;: print x
110 print using a$; y;: print y
120 print using a$; z;: print z
Questo produce:
3333333e-07  .333333333
1234000e-03  1234
9998990e+03  9.99899e+09
10 pudef"*.,L"
```

5.2 PUDEF

L'istruzione PUDEF è così piccola e semplice da sembrare una sciocchezza. Permette di dire al calcolatore di stampare un altro carattere al posto di uno dei seguenti:

spazio   virgola   punto   dollaro

Prende dunque la forma:

PUDEF " ,.$"

In questa forma non cambierebbe niente. Ma PUDEF "*,.$", stamperà asterischi in tutti gli spazi, mentre PUDEF " .,£" sostituirà al dollaro il segno £. Analogamente, PUDEF " .,$" stamperà punti al posto delle virgole e virgole al posto dei punti. Questa possibilità non è trascurabile, poiché i risultati di un calcolo eseguito con il C-128 hanno il punto dove l'usanza italiana richiede la virgola.
Attenzione: come giustamente fa presente il Manuale, se scriviamo per esempio PUDEF " £", allora la macchina scriverà spazi al posto dei punti e delle virgole. Se vogliamo conservare spazio, punto e virgola come tali, dobbiamo scrivere PUDEF " ,.£".
Invece, se volessimo sostituire soltanto lo spazio, per esempio con 0, allora basterebbe l'istruzione: PUDEF "0".
Torniamo al primo programma descritto in questo capitolo e aggiungiamo la riga:

```
10 pudef"*.,L"
```

Produce le seguenti righe di stampa, identiche a quelle prodotte dal primo programma, ma con * al posto degli spazi, con virgole e punti scambiati e con L al posto di $:

```
****1,23450
12345,00000
12345,67890
****1,2
12345,0
12345,7
 +****1,23
 +12345,00
 +12345,68
 $*****1
 $*12345
 $*12346
 ******L1,23
 **L12345,00
 **L12345,68
```

Questo mostra che (se per esempio si ha una stampante che non dispone del simbolo £ per Lira), si può usare un altro carattere. Nel quarto gruppo di cifre il $ è rimasto tale e quale: infatti non è preceduto dal alcun #. Di conseguenza viene trattato come carattere "letterale": cioè né PRINT USING né PUDEF lo sottopongono ad alcun trattamento speciale: qualsiasi carattere in quella posizione sarebbe trattato letteralmente; al contrario, le righe 500..., nelle quali è preceduto da un #, lo vedono come parte del formato e lo attaccano alla prima posizione del valore numerico. Il programma sul nostro disco PUDEF fornisce altri esempi.

5.3 CHAR

Per concludere, un altro modo di stampare caratteri sullo schermo. CHAR ha in realtà lo scopo principale di consentire al programmatore di scrivere caratteri sullo schermo grafico, non accessibile a un normale PRINT. Ciò nonostante, può essere utilissimo sullo schermo normale. Oltre a dire che cosa si deve stampare, specifica anche la posizione, il colore, e così via. Ha la sintassi:

CHAR [*fonte colore*], *x*, *y* [,*stringa*] [,*rvs*]

fonte colore è un numero da 0 a 3 (si veda COLOR nella System Guide). Se si omette un numero qui, il colore sarà quello attualmente definito con COLOR 1. Attenzione al fatto che la virgola è necessaria anche se non si vuole specificare il colore.

x, *y* sono le coordinate: x=colonna (da 0 a 79; sullo schermo a 40 colonne un numero superiore a 39 farà iniziare la stringa sulla riga successiva); y=riga (da 0 a 24 su entrambi gli schermi).

,stringa può essere un gruppo di caratteri tra virgolette, oppure una variabile (come a$).

,rvs (reverse) può essere 0 o 1: 1 fa apparire la stringa in negativo; 0 (o anche niente, perché è il default) la stampa normalmente.

Altre informazioni ed esempi sull'uso di CHAR in modo grafico sono forniti nei capitoli dedicati alla grafica. Sul disco esiste un breve tutorial. Il grande pregio di CHAR è la possibilità di definire *x* e *y*. In questo senso è utilissimo anche sullo schermo non grafico. Infatti possiamo controllare queste coordinate tramite variabili numeriche, stabilendo con assoluta precisione le posizioni delle stringhe anche quando il contenuto dello schermo è altamente variabile. Questo lo rende prezioso nella compilazione di tabelle sullo schermo. Sullo schermo non grafico, CHAR ignora la fonte specificata.

```
CHAR 1, 10, 15, "xyz", 1
```

Sullo schermo grafico stamperà quindi "xyz" in negativo e nel colore precedentemente definito con COLOR 1, a partire dalla decima colonna della sedicesima riga dello schermo (il numero della prima riga è 0). Sullo schermo di testo, invece, il colore (fonte 1 nell'esempio) viene ignorato: "xyz" apparirà nel colore stabilito per il testo (fonte 5). Il parametro *rvs*=1 continua invece ad assicurare il negativo. Si possono usare i soliti caratteri di controllo (con CONTROL e COMMODORE) per stabilire i colori. Ciò non è possibile se si desidera usare lo stesso comando CHAR sia sullo

schermo grafico che sullo schermo di testo: sullo schermo grafico infatti non modificano il colore dei caratteri ma appaiono i simboli stessi (\ per il rosso, ecc. Si veda anche il tutorial CHAR sul disco). Un'altra differenza di comportamento sui due tipi di schermi consiste nel modo di usare maiuscolo e minuscolo. Sullo schermo normale, CHAR funziona esattamente come PRINT, e quindi se si sta usando il modo "maiuscolo e minuscolo" (modo CHR$(14)), anche CHAR produce maiuscolo e minuscolo a seconda che sia premuto o meno il tasto SHIFT. Sullo schermo grafico, invece, il default è maiuscolo e caratteri grafici (modo CHR$(142)), e se vogliamo maiuscolo e minuscolo, ogni singola istruzione CHAR lo deve indicare:

```
CHAR 1, 3, 7, CHR$(14)+"Sopra la panca la capra campa."
CHAR 1, 3, 8, CHR$(14)+"Sotto la panca, invece..."
```

Se non indichiamo CHR$(14) in questa forma tutte le volte, al posto delle maiuscole avremo caratteri grafici.
CHAR, infine, è molto più facile da usare, rispetto a PRINT, se si vuole presentare sullo schermo un dato (per esempio un valore numerico) che cambia rapidamente. Diversi programmi usano questa possibilità: un buon esempio si trova nel tutorial SOUND, e un altro nello stesso tutorial CHAR.

6

INSTR e altre funzioni per le stringhe

L'uso di classificare le varie voci del BASIC nei vari tipi (comando, istruzione o specifica, funzione) ha sempre meno significato, e infatti questo libro non si preoccupa eccessivamente delle ormai teoriche differenze. Il termine funzione è l'unico che ha una certa utilità nella classificazione delle voci del BASIC. Anche così, le funzioni potrebbero essere ulteriormente divise: esistono funzioni (come HEX$ oppure SQR) che richiedono in pratica l'esecuzione di un calcolo da parte del sistema; altre contengono informazioni sulle condizioni operative del calcolatore (come ER oppure DS$).
La System Guide separa le funzioni dalle altre voci del BASIC. Questo è logico in quanto il termine funzione è l'unico che conserva un significato utile, illogico in quanto non è sempre evidente per il lettore che una funzione sia tale. Per esempio, dal punto di vista di chi non conosce il BASIC, non è chiaro perché WINDOW viene classificato come comando, mentre TAB oppure SPC sono chiamate funzioni. Questo significa che l'utente può essere costretto a cercare una voce due volte nelle due sezioni, e che sfogliando il manuale d'uso le funzioni gli capiteranno meno spesso sotto gli occhi perché questa sezione è molto più piccola dell'altra.
Il BASIC 7.0 ha una serie di funzioni nuove, alcune molto utili, altre di meno evidente utilità. HEX$ e DEC (v. dischetto) consentono la conversione tra numerazione esadecimale e decimale. ER, EL, ERR$, DS, e DS$ danno informazioni chiare sugli errori. BUMP, JOY, PEN, POT, riflettono i risultati di informazioni fornite dall'utente. POINTER, RCLR, RDOT, RGR, RSPCOLOR, RSPPOS, RSPRITE, RWINDOW informano sulle attuali condizioni operative del sistema. XOR offre l'unica funzione booleana nuova rispetto al C-64, con una modalità d'uso diversa dagli altri operatori logici come

AND, OR, NOT, ecc. Tra le funzioni applicate alle stringhe, che sono importantissime, l'unica novità è INSTR. Ma il suo valore è immenso: completa la gamma che comincia con LEFT$, RIGHT$ e MID$ e che consente in pratica di usare il calcolatore per esaminare parole e frasi.
Questo capitolo offre un breve riesame di queste funzioni, comprese quelle non nuove, per illustrare la maggior potenza conferita al BASIC dalla presenza di INSTR. Accenna poi ai problemi connessi con l'uso di INPUT ed esamina altri problemi che si riscontrano nella revisione di programmi lunghi.

6.1 INSTR

Rende veramente facile tutta una serie di operazioni che prima erano da certosini. Supponiamo di avere la stringa A$="La vispa Teresa". Vogliamo che il calcolatore controlli una serie di stringhe, e ne selezioni una che parli appunto di Teresa. Da una parte non siamo in grado di prevedere tutte le stringhe che l'utente può immettere; dall'altra non vorremmo scrivere in memoria tutte le possibilità, anche se fossero prevedibili. Allora si può usare questa funzione:

```
100 if instr(a$,"Teresa") then print "Ti piace, eh?"
```

Questa linea di programma è uguale a

```
100 if instr(a$,"Teresa")<>0 then...
```

Ovviamente anche "Teresa" può essere contenuto in una variabile: ne potremmo avere un gran numero e, usando INSTR in un loop, far reagire il programma in mille modi diversi a seconda dei nomi. Più o meno così:

```
10 input "Scrivimi una frase"; n$: trap 2000
20 do
30 :   read a$
40 loop until instr(n$,a$)
50 print "Anche a me piace la "a$
60 end
100 data Maria, Anna, Giovanna, Susanna
110 data Teresa, Filiberta, Zia, Mamma
2000 print "Una frase con il nome di una donna che conosco":
     sleep 10: run
```

Questo programmino, stupidissimo, serve a illustrare il concetto elementare di INSTR. L'INPUT chiede una frase: scriviamo "La vispa Teresa..."

oppure "Cara dolce Mamma..." e (se la frase contiene un nome di donna tra quelli nell'istruzione DATA) il computer dice che anche a lui piace Teresa o la Mamma... In caso contrario, l'esaurimento dei DATA disponibili provoca un errore, che viene intercettato da TRAP. La riga 2000 quindi dice all'utente di usare un nome tra quelli nelle righe DATA, e ridà il RUN al programma.
Il lettore dovrebbe conoscere già le funzioni LEFT$, RIGHT$ e MID$. INSTR ne completa le possibilità. Vediamo prima la sintassi di INSTR:

INSTR (*stringa1, stringa2* [*,iniz*])

Cerca, come abbiamo visto, *stringa 2* all'interno di *stringa 1*, a partire da *,iniz* se questo è indicato. Se lo omettiamo, il calcolatore assume 1 (primo carattere a sinistra in *stringa 1*).
Se abbiamo una stringa A$="1234567890ABCDEFG":

PRINT INSTR(A$,"A") produce 11
PRINT INSTR(A$,"A",11) produce pure 11, mentre
PRINT INSTR(A$,"A",12) oppure
PRINT INSTR(A$,"Z")

producono entrambi 0, perché non c'è una A oltre l'undicesima posizione, e perché la Z non esiste affatto.
Per riassumere: il parametro *iniz* fa considerare solo i caratteri a partire dalla posizione stessa. Il valore numerico prodotto da INSTR, invece, parte sempre dall'inizio di *stringa 1*. Per analizzare le stringhe, abbiamo a disposizione quindi varie funzioni:

ASC - per esempio per sapere se un carattere è maiuscolo
LEFT$, MID$, RIGHT$ - per estrarre dei pezzi da una stringa
LEN - per sapere la lunghezza della stringa
INSTR - per cercare una variabile stringa dentro la stringa
STR$ - per cercare una variabile numerica dentro la stringa
CHR$ - per cercare un determinato carattere in una stringa.

In alternativa si può generalmente usare anche il carattere in questione tra virgolette, ma se vogliamo cercare prima la A, poi la B... e così via, servendoci di un loop (FOR T=32 TO 128: IF INSTR(A$,CHR$(T))=...), CHR$ è preziosa in questa come in molte altre applicazioni.
Evidentemente, queste funzioni hanno anche altri usi: ciò che ci interessa in questo momento è però la loro utilità nell'esame di una serie di stringhe per determinarne il contenuto.
Ho usato alcuni fra questi per compilare un altro piccolo programma, che è l'esempio compreso sul disco:

```
20 dim a$(128)
30 input"Scrivere una frase";a$
40 :   if instr(a$,chr$(32)) then 50:
       else print "Questa non e´ una frase":goto30
50 :   if instr(a$,".") = 0 then
       print "Non finisce con punto... ma OK"
60 :   if asc(a$) < asc("A") then
       print "Non comincia con la maiuscola... ma OK"
65 print "Premere un tasto": getkey o$
70 l = len(a$)
80 x = 1: y = 1: i = 1
90 do
100 :  i = instr(a$," ",x)
110 :  a$(y) = mid$(a$,x,i-x)
120 :  x = i + 1: y = y + 1
130 :  loop until instr(a$," ",x) = 0
140 a$(y) = mid$(a$,i + 1,len(a$) - i)
150 print "La frase contiene"; y; "parole": sleep2
160 for t = 1 to y: print a$(t),tab(8): next
165 print: print "Premere un tasto": getkey o$
```

Molto primitivo... ma illustra il tipo di operazioni che si possono compiere sulle stringhe. Le righe 40 e 50 cercano rispettivamente la presenza di spazi e del punto. Se non ci sono spazi il programma, sdegnato, si rifiuta di funzionare: se non c'è il punto si limita a brontolare. La funzione ASC nella 60 controlla il valore del primo carattere: se non è maiuscolo brontola di nuovo.
Il loop (90-130) prende tutti i caratteri che ci sono tra due spazi (o tra l'inizio della stringa e il primo spazio); la 120 incrementa x in modo che corrisponda al primo carattere dopo lo spazio, e y per aumentare di 1 l'indice di A$(*y*), creando così tante stringhe di una parola quante sono le parole in A$. Nella sua forma attuale, questo programmino non servirebbe a fare un lavoro serio, ma illustra benissimo il tipo di filosofia che sta alla base di molte operazioni di elaborazione di testi.
I vantaggi della nuova funzione sono quelli tipici di questo nuovo BASIC: INSTR, più che dare nuove possibilità, risparmia molto lavoro e soprattutto rende più leggibile (e quindi più facilmente controllabile) il programma. Nulla infatti ci impedisce di usare MID$ per lo stesso lavoro, ma chi lo preferisse sarebbe masochista. Non so se INSTR sia più veloce: bisognerebbe compilare due programmi molto lunghi usando molte volte le due possibilità, perché altrimenti la differenza di tempo (eventuale) non sarebbe percepibile. Ma se noi consideriamo che chi lavora sui testi non ha generalmente le grandi esigenze di velocità che si possono invece avere con i videogiochi, la questione è più che altro d'interesse accademico.
Sarebbe errato trascurare voci del BASIC utili quanto MID$ e ASC. La prima, come abbiamo già visto, è utilissima anche sul C-128, dove per

un'operazione è stata soppiantata da INSTR. MID$ è illustrato bene nella versione in BASIC del programma DIRTUT (Lato 1 del dischetto, a partire dalla riga 640).
Un fatto nuovo è che MID$ può essere usato come istruzione, per sostituire caratteri in una stringa. Questa possibilità, ignorata nella documentazione Commodore, è ampiamente documentata nel Dizionario del BASIC.
Ora che disponiamo di INSTR, MID$ non serve più tanto a scoprire se una stringa contiene una determinata cosa, quanto a lavorare su quella cosa quando c'è. INSTR, infatti, è solo informativo, e non ci consente di estrarre una parte di una stringa per crearne una nuova, ma ci consente di trovare il valore di x (ed eventualmente di y) nell'espressione:

MID$(A$, *x*, *y*)

dove x è il numero (a partire da sinistra) del primo carattere da considerare, e y è il numero dei caratteri da prendere in considerazione a partire da x.

LEFT$(A$, *x*) e RIGHT$(A$, *x*)

considerano i primi x caratteri, a partire rispettivamente da sinistra e da destra, nella stringa A$. In certe operazioni sono più comode, ma MID$ può essere usata quasi sempre.
ASC(A$) dà il numero ASCII (CHR$) del primo carattere della stringa A$, e non considera gli altri. È utile nel riconoscimento di stringhe che cominciano con un determinato carattere, per esempio nell'ordinamento alfabetico, oppure per controllare che il primo carattere non sia lo spazio (CHR$(32)):

```
IF ASC(A$)=32 THEN A$=RIGHT$(A$, LEN(A$)-1)
```

elimina tali spazi, che sono abbastanza comuni (anche se un INPUT, se riceve dall'utente una stringa che comincia con uno spazio, lo ignora). Un caso tipico è la stringa ottenuta con STR$, che se rappresenta un numero positivo ha uno spazio nella posizione 1. L'espressione per RIGHT$ dice infatti di considerare tutto A$ meno il primo carattere.
Potremmo usare ASC per stabilire, per esempio, se una parola ottenuta con il programmino INSTR-TEST 1 sia un nome proprio.

```
IF ASC(A$)>192 THEN...
```

6.2 ALFASORT

Non trascuriamo il fatto che, nell'ordinamento alfabetico di una serie di parole, decidere quale stringa viene prima nell'ordine alfabetico è molto facile:

IF A$ >B$ THEN PRINT A$: ELSE PRINT B$...

Anche questa operazione considera il valore ASCII delle stringhe, ma se il primo carattere è uguale nelle due stringhe, considera anche il secondo, e così via. Si può riempire una matrice di stringhe da mettere in ordine alfabetico e selezionarle in ordine alfabetico senza altre formalità. Il principio fondamentale è abbastanza semplice. Il programma ALFASORT è un programma piuttosto primitivo, ma è servito per ordinare l'indice analitico di questo libro. Il principio è illustrato nelle righe seguenti. Ovviamente la stringa A$ va dimensionata: la dimensione dipende in teoria solo dalla capacità della memoria del calcolatore, che è abbondante, ma se le stringhe da selezionare sono numerose, la versione in BASIC diventa lentissima. Per questo motivo la versione usata è stata scritta in BASIC e poi compilata in linguaggio macchina, che ha consentito una velocità 5 volte maggiore. Per motivi di spazio la versione compilata non è presente sul disco. Il principio è illustrato nel seguente estratto:

```
160 for i = 1 to n
170 :  print i;: input a$(i)
180 next i
190 if rgr(0) = 0 then print "Attendere": sleep 1
200 fast
210 for i = 1 to n-1
220 :   if a$(i+1) > = a$(i) then 270
230 :   b$ = a$(i+1)
240 :   a$(i+1) = a$(i)
250 :   a$(i) = b$
260 :   goto 210
270 next i
280 slow
290 print ""
300 for i = 1 to n
310 :   print a$(i)
320 :   if i/20 = int(i/20) then print "Premere un tasto":
        getkey a$
330 next i
```

Questo è un programma di sort semplice e lento, uno dei casi in cui FAST diventa utile, anche se significa non vedere niente sullo schermo a 40 colonne. Anche così, il tempo richiesto aumenta vertiginosamente con

il numero delle stringhe da selezionare: infatti, ogni volta che viene spostata una stringa, il loop ricomincia da capo. Si può risparmiare del tempo ordinando dei piccoli gruppi di stringhe e poi cucendole insieme con il word processor, per ottenere un file parzialmente già in ordine. Ma anche così ordinare alfabeticamente l'indice di questo volume ha richiesto diverse ore. Fortunatamente esistono sistemi più veloci.
La riga 320 serve soltanto a fermare la stampa ogni volta che lo schermo si riempie.

6.3 VAL e STR$

La funzione VAL legge un valore in una stringa, purché questo preceda gli eventuali caratteri non numerici. Fa il contrario di STR$, che converte un numero in stringa.

```
X +VAL(A$)
X$=STR$(X)
```

Tenere presente lo spazio che risulterà all'inizio di X$ se X è un valore positivo.

```
X$="500 FIAT": Y$="FIAT 500"
PRINT VAL(X$); VAL(Y$)
  500 0
```

6.4 Nomi delle variabili

Il problema più frequente che si incontra scrivendo programmi lunghi e complessi, è che non si scelgono abbastanza sistematicamente i nomi delle variabili: poi si dimenticano i vari significati e si duplicano variabili già esistenti. Una soluzione è quella di usare nomi lunghi. Anche se il BASIC 7.0 considera solo i primi due caratteri, il nome può avere fino a 254 caratteri. TIME$ e TI$ sono la stessa cosa, ma TIME$ è più leggibile. Potremmo quindi usare NOME$ invece di NO$, INDIRIZZO$ al posto di IN$, e via dicendo. Esiste solo il rischio di incorporare nel nome i caratteri riservati a qualche comando, istruzione, funzione o variabile del sistema. Infatti, questi nomi non possono essere usati in un nome di variabile, anche insieme ad altri caratteri. Così STIPENDIO è vietato perché contiene sia ST che TI, e così via.

7

Il bit mapping: GRAPHIC e SCALE

Dopo una discussione dei concetti che governano la riproduzione delle immagini sullo schermo, questo capitolo esamina i vari schermi disponibili sul C-128. Le voci del BASIC discusse sono GRAPHIC e SCALE.
La produzione di una forma sullo schermo è sempre una operazione grafica, anche quando lavoriamo con lo schermo "normale", dove le forme in questione sono i caratteri (lettere, numeri o i cosiddetti caratteri grafici), queste operazioni si servono dei dati scritti permanentemente nella ROM (Read-Only Memory). Le informazioni grafiche necessarie per scrivere caratteri sono quindi sempre presenti.
Con i caratteri grafici disponibili dalla tastiera è possibile creare composizioni interessanti, ma evidentemente le possibilità sono limitate dal fatto che i caratteri in questione sono sempre gli stessi. È possibile sostituirli, tramite istruzioni che inducono il calcolatore a cercare le informazioni in un'altra area di memoria (RAM), dove l'utente può inserire qualsiasi forma. Praticamente tutti i libri che sono stati scritti sul C-64 descrivono il metodo da seguire.

7.1 Bit, byte, pixel e sistemi numerici

Le unità fondamentali per la gestione dei dati per qualsiasi scopo nella memoria di ogni calcolatore sono due: il bit e il byte. Il bit è un solo numero, e può essere al minimo 0 e al massimo 1. Con un solo bit, non si possono trattare valori superiori a 1 o numeri negativi.
Quando usiamo i normali numeri decimali, le unità fondamentali sono 10

(0-9). Non ci preoccupiamo in genere di tener presente che la posizione di una cifra nel numero ne indica il valore come potenza di 10:

$$
\begin{aligned}
1 &= 10^0 \\
11 &= 10^1 + 10^0 \\
111 &= 10^2 + 10^1 + 10^0 \\
1234 &= 10^3 + (2 \times 10^2) \times (3 \times 10^1) + (4 \times 10^0) \\
90000000 &= (9 \times 10^6)
\end{aligned}
$$

Lavorando "in base 2", invece, le unità fondamentali sono solo 2, e le colonne a partire da destra corrispondono a 2^0, 2^1, 2^2, 2^3, 2^4 ...

1	=	2^0	1 decimale
10	=	2^1	2
11	=	$2^1 + 2^0$	3
100	=	2^2	4
101	=	$2^2 + 2^0$	5
110	=	$2^2 + 2^1$	6
111	=	$2^2 + 2^1 + 2^0$	7
1000	=	2^3	8

Così, il numero massimo che possiamo scrivere con 4 cifre in codice binario è

$$1111 = 2^3 + 2^2 + 2^1 + 2^0 \quad (15)$$

Il calcolatore organizza i bit a byte, gruppi fissi di cifre binarie che normalmente hanno 8 cifre (bit) ciascuna. Il bit può avere il valore 1 oppure 0: ciò significa che il valore numerico più alto che si può esprimere con un solo byte è 255

$$11111111 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$$

Non deve sfuggire che, dal momento che 0 è un valore numerico, un byte può esprimere 256 valori numerici diversi.
Questo valore di 256 rappresenta anche il quadrato di 16 (16^2). I calcolatori si servono quindi per molte operazioni della numerazione binaria (base 2), o di quella esadecimale (base 16) anziché della numerazione decimale (base 10) a noi più familiare.
Se tutto questo sembra complesso, basta pensare che neanche gli esseri umani usano soltanto il sistema "a base 10". L'esempio più ovvio è il nostro modo di misurare il tempo. La giornata è divisa in 2 periodi di 12 ore: le ore sono divise in 60 minuti e i minuti in 60 secondi. La nostra base di calcolo, in questa che è forse l'operazione di aritmetica mentale che facciamo più spesso, è quindi addirittura triplice. È a base 2-12-60, men-

tre il calcolatore usa un sistema a base 2-16 soltanto. E non è un sistema antiquato come certe unità di misura americane (il piede, la pinta, e così via): lo usiamo perché è comodo ed efficiente. Mentre 10 e 100 sono scomodi (per esempio non sono divisibili per 3) $2\times12=24$ è già divisibile per 2,3,4,6,8. E $2\times12\times60=1440$ è divisibile per 2,3,4,5,6,8,9,10,12.
Noi forse usiamo più spesso il sistema a base 10 perché abbiamo un byte portatile (le nostre 2 mani con 10 dita). La mano del C-128 è il byte: questo normalmente ha 8 "dita" (bit), ma per certe operazioni si lavora con byte di 4 bit (detti "nibble") e in alcuni casi il byte può essere di 12 bit. Avere in memoria un valore numerico significa quindi dedicare a quel valore tanti byte quanti, messi insieme, sono necessari a esprimerlo. Per esempio se un byte può rappresentare un valore fino a 256, con 2 byte possiamo scrivere un valore fino a $256\times256=65536$.
Rappresentare un valore numerico nella memoria del calcolatore, in una forma tale che la macchina riesca a compiere delle operazioni, non equivale però a scrivere il numero sullo schermo. Si tratta di rappresentare non il valore del carattere ma la forma. Per avere un buon grado di leggibilità è necessario dedicare a ciascun carattere sullo schermo una matrice di punti di almeno 8×8, cioè di 8 byte o di 64 bit.
In questo modo possiamo rappresentare la lettera A con i seguenti bit:

```
 24  00011000     **     $0018
 36  00100100    *  *    $0024
 66  01000010   *    *   $0042
126  01111110   ******   $007E
 66  01000010   *    *   $0042
 66  01000010   *    *   $0042
 66  01000010   *    *   $0042
 00  00000000            $0000
```

A sinistra i numeri decimali, e a destra i numeri esadecimali, indicano il valore contenuto in ciascuno degli 8 byte; la seconda colonna ne dà gli equivalenti in codice binario. La A fatta di asterischi rende più visibile l'effetto, peraltro riconoscibile nelle posizioni degli 1 fra i numeri binari. Il programma BINARIO (Lato 1, Menu Comandi Vari) consente di tradurre tra decimale e binario e viceversa e di visualizzare gruppi di 8 byte in codice binario come nell'esempio riportato qui. Fornisce anche i corrispondenti valori esadecimali. Notare che per esprimere un numero esadecimale si usano non solo le cifre 0-1-2-3-4-5-6-7-8-9 ma anche le "cifre" A-B-C-D-E-F, corrispondenti ai valori decimali 10-11-12-13-14-15. Il numero esadecimale $10 corrisponde quindi a 16 in notazione decimale, mentre $FF=255 e $100=256. Si veda il programma HEX/DEC.
Per riprodurre un carattere sullo schermo, un solo numero (codice di schermo: Appendice F della System Guide) definisce quale carattere è da

riprodurre (@=0, A=1, e così via fino a 255, per un totale di 256 caratteri). Questo codice è caratteristico dei calcolatori Commodore e un altro calcolatore non lo capirebbe. Tutti gli altri calcolatori delle più diverse marche usano invece il codice ASCII (American Standard Code for Information Interchange).
In base al codice di schermo il C-128 sa dove cercare nella sua memoria gli 8 byte che rappresentano la forma grafica del carattere. È un indirizzo e serve soltanto a questa operazione: ci risparmia la necessità di inserire nel programma tutti gli 8 valori, tutte le volte, consentendo al calcolatore di farlo per noi.
Nella parte della memoria del calcolatore che lavora invece sul significato (valore alfabetico) dei caratteri, ogni carattere è rappresentato da un solo byte contenente un codice ASCII. In questo codice il valore della lettera A, per esempio, è 65: se eseguiamo un PEEK in una posizione di memoria che contiene una A, riceveremo questo numero come risposta. La funzione CHR$(*n*) è un modo di rappresentare il valore ASCII di un carattere: così PRINT "A" e PRINT CHR$(65) produrranno lo stesso effetto.
Per tornare alla forma grafica del carattere, è necessario ricordare che lo schermo è coperto di punti, ognuno dei quali può essere acceso (1) o spento (0). Ogni punto viene chiamato pixel (*picture cell*) e corrisponde appunto a un bit nello stato 1 o 0. SPENTO significa che il pixel ha il colore dello sfondo e non si vede: ACCESO vuole dire che ha un altro colore e quindi è visibile. La pressione di un tasto, o il comando PRINT, usano le immagini predeterminate nella ROM per produrre un carattere sullo schermo.
I limiti imposti da questo metodo di formare immagini sullo schermo sono accettabili per molti scopi, ma se vogliamo creare sullo schermo immagini diverse da quelle consentite usando i caratteri contenuti nella ROM, dobbiamo usare il modo GRAPHIC del C-128.
Questo, ben più complesso e dispendioso per la quantità di memoria che consuma, dimentica il byte come unità di lavoro e divide lo schermo in tanti pixel, direttamente. Per ottenere questo risultato il C-128 deve controllare indipendentemente ognuno dei 64000 pixel sullo schermo, e questo richiede 8 K di memoria. E non basta: un controllo viene esercitato anche sui 16 colori possibili. Il colore (ed è una limitazione) continua a essere gestito a gruppi di 8 byte: cioè lo schermo è diviso in 1000 quadretti, ognuno dei quali corrisponde a un colore. La grafica multicolor aggira in parte questo ostacolo, ma rimane una notevole limitazione.
La disponibilità di punti effettiva è quindi di 64000 pixel, numerati (se non se ne modifica la numerazione con il comando SCALE) con coordinate da 0 a 319 e da 0 a 199, rispettivamente per l'asse orizzontale (x) e per l'asse verticale (y).
Per cambiare le modalità di funzionamento dello schermo si usa il comando GRAPHIC.

7.2 GRAPHIC 0

Quando si accende il calcolatore, se non è abbassato il tasto 40/80 DISPLAY si è nel modo GRAPHIC 0, cioè non esiste nessuno schermo grafico. In questa modalità di funzionamento i programmi in BASIC sono collocati all'inizio della zona di memoria riservata al BASIC, che parte da 7168 ($1C00) e finisce a 64511 ($FBFF). Se si cambia modo grafico, il BASIC viene spostato per riservare alla grafica un'area di memoria di 9 kbyte. Questa inizia all'indirizzo 7168 e finisce a 16383 (da $1C00 a $3FFF in codice esadecimale). In questo modo il BASIC inizia quindi a 16384 ($4000). Data l'ampiezza della memoria del 128, questo è di regola accettabile, ma se ci serve ricuperare lo spazio per usarlo nello stesso programma (o in un altro, senza dover spegnere e riaccendere il calcolatore), il comando è GRAPHIC CLR. Il programma viene "riallocato", cioè torna ad occupare tutta la memoria normalmente disponibile per il BASIC.
Sono 3 i tipi principali di schermo disponibili:

- 0 lo schermo di testo a 40 colonne
- 1-4 grafica mono e multicolor
- 5 lo schermo a 80 colonne (si veda il capitolo sullo schermo ad 80 colonne).

Il tipo che ci interessa qui è definito dai numeri 1-4.

7.3 GRAPHIC 1 - Grafica ad alta risoluzione

Questo comando (da tastiera o da programma) ha l'effetto di far scomparire lo schermo a 40. Se lo usiamo così, la prima volta dopo l'accensione del calcolatore, lo schermo apparirà pieno di "rifiuti" (caratteri e colori senza senso). Per avere uno schermo pulito bisogna aggiungere l'argomento:

```
GRAPHIC 1,1
```

Questo ha circa lo stesso effetto del tasto CLR sullo schermo normale. Elimina tutte le scritte, ecc., presenti sullo schermo, che resta di conseguenza totalmente vuoto. È importante rendersi conto dell'effetto che si ottiene con questo argomento. Se si omette, si evita che certi cambiamenti effettuati per esempio sullo schermo di testo a 40 colonne si ripercuotano sullo schermo grafico. Per esempio, se inseriamo

```
COLOR 0, 3: GRAPHIC 1, 1
```

il primo comando colora di rosso lo sfondo dello schermo a 40 colonne, e il secondo porta questo colore rosso anche allo schermo grafico. Se poi eseguiamo:

GRAPHIC 0: COLOR 0, 16: GRAPHIC 1

lo schermo di testo diventa grigio, ma l'assenza del secondo 1 nel comando GRAPHIC protegge lo schermo grafico, il cui sfondo rimane rosso. È naturalmente importante rendersi conto di questa possibilità.

7.4 GRAPHIC 2

È uno schermo che ha le stesse proprietà di GRAPHIC 1, con l'aggiunta di una terza opzione. La sintassi è quindi:

GRAPHIC 2, *clr*, *n*

La terza opzione apre una finestra sullo schermo normale a 40 colonne, a partire dalla *n*-esima riga dello schermo. In questa finestra si possono vedere righe di testo normale mentre nella parte superiore si vedono grafici. Il default per *n* (cioè il valore che si ottiene se non si indica alcun numero) è 19, poiché le righe si numerano a partire da 0 e sono 25 in totale, questo significa che si avrà una finestra di 5 righe a partire dalla ventesima riga. Basta variare in più o in meno questo numero per avere più o meno spazio a disposizione. È una possibilità utilissima, sia per studiare un grafico in modo diretto, sia per collaudare un programma, sia anche per presentare commenti o istruzioni all'utente durante il normale svolgimento del programma, senza usare il comando CHAR per scrivere i caratteri direttamente sullo schermo grafico.
Si può cambiare tranquillamente tra GRAPHIC 0, 1, 2, 3 e 4 senza interferire con il contenuto dello schermo, a patto di NON usare l'opzione *clr*. GRAPHIC 1 e GRAPHIC 3 danno una strana possibilità che può essere utile. Lo schermo di testo continua ad essere attivo, anche se è invisibile. Questo significa che quando si torna in modo GRAPHIC 0 si possono trovare indicazioni stampate mentre lo schermo non era visibile. Se invece si era in modo GRAPHIC 5 (80 colonne), il segnale per lo schermo RGBI a 80 colonne resta disponibile, e se abbiamo un monitor RGBI oltre a un monitor normale o una TV, è possibile usare entrambi. In questo modo diventa piuttosto facile disegnare grafici e modificare il programma senza dover entrare e uscire dal modo grafico. Quando si lavora con lo schermo grafico nei modi GRAPHIC 1 o GRAPHIC 3, la funzione RGR dà valori (rispettivamente 6 e 8). Questi corrispondono alla somma dei due

modi grafici in vigore. È un'altra informazione non riportata nella System Guide. Se non si dispone dei due schermi ci si deve accontentare delle possibilità offerte da GRAPHIC 2 e da GRAPHIC 4, con la finestra sullo schermo grafico.

7.5 GRAPHIC 3 e 4 - Grafica a bassa risoluzione

In questi due modi (che in realtà sono uno solo, perché GRAPHIC 4 è, come già abbiamo visto per GRAPHIC 2, semplicemente una versione con una finestra in basso), abbiamo altri due colori disponibili.
Il programma tutorial MULTICOLOR dà un'idea generale delle differenze: si veda invece il programma MULTIDRAW per un'applicazione pratica. In modo multicolor si usa COLOR 2 e COLOR 3 per definire i due colori addizionali disponibili: in modo grafico normale queste fonti non sono disponibili, e se si tenta di usarle viene segnalato un errore.
Il vantaggio principale del modo multicolor consiste nel fatto che, poiché si dispone di 4 fonti e non di due, è più facile evitare che i colori "spandano" come accade in GRAPHIC 1/2 (come si è detto, mentre i pixel si disegnano bit-per-bit, i colori vanno byte-per-byte per ciascuna fonte). È però indispensabile evitare di usare la stessa fonte (o sorgente) di colore con colori diversi all'interno dello stesso byte. Naturalmente si può disegnare per esempio con fonte 3 e colorare con fonte 1, e così via. Il programma MULTIDRAW è studiato per abituare il programmatore a prevedere gli errori che si possono fare nella scelta delle fonti, in modo da evitare gli effetti che si ottengono quando i colori "spandono". In modo GRAPHIC 1/2 deve esistere una rigida disciplina, perché due colori che vengono a contatto l'uno con l'altro producono quasi sempre questo effetto. Ma la limitazione può anche tornare utile: è possibile usare PAINT per colorare tutto lo schermo usando una fonte qualsiasi: se poi si cambia il colore contenuto nella stessa fonte, e si usa questa per disegnare, si ottengono disegni dal tratto molto largo (8 pixel). La maggiore accuratezza dei colori in GRAPHIC 3/4 si paga con una minore risoluzione orizzontale (160 anziché 320 pixel).
Le differenze risultano più chiare tramite i programmi del dischetto che con una spiegazione scritta. In breve:

- GRAPHIC 1/2 dà linee più nitide (specie le curve e le linee diagonali). Ma nel programma ELLISSE il cerchio interno è danneggiato dal PAINT, perché vengono riempiti byte interi di colore, e quindi alla fine del programma il cerchio centrale diventa un gruppetto informe di byte senza più la forma di un cerchio.
- GRAPHIC 3/4 dà righe meno nitide perché sull'asse x 2 pixel corrispondono a un solo bit, e c'è quindi una definizione minore in direzio-

ne orizzontale. In compenso si conserva la forma del cerchio interno. Non si può avere la botte piena e la moglie ubriaca.

- Infine, mentre l'opzione *,clr,* se omessa, non permette alcun cambiamento al grafico in modo GRAPHIC 1/2, in modo 3/4 si può cambiare il colore dello sfondo (COLOR 0,*n* cambia tutti i pixel colorati nella fonte 0 in qualsiasi momento: in altre parole lo schermo in bassa risoluzione di GRAPHIC 3/4 ha sempre lo stesso colore di sfondo dello schermo di testo).

7.6 GRAPHIC 5

Questa, la sesta ed ultima possibilità della serie GRAPHIC, consente di passare allo schermo a 80 colonne. Si veda il Capitolo 9 dedicato agli schermi di testo.

7.7 SCALE

SCALE consente di cambiare la scala numerica con la quale si definiscono i pixel.
I pixel sono normalmente definiti sulla scala reale (che è il default), per la quale

x=0-319 (totale di 320 punti orizzontali nei modi GRAPHIC 1/2)
x=0-160 (totale di 160 punti orizzontali nei modi GRAPHIC 3/4)
y=0-199 (totale di 200 punti verticali)

Questa è la scala che si ottiene:

a) se non si usa il comando SCALE dopo i comandi GRAPHIC 1/2/3/4;
b) se si usa il comando SCALE 0

È importante capire che usare una scala diversa dal default NON significa aumentare i numeri dei pixel realmente presenti sullo schermo. Poiché SCALE può andare da 0 a 65535 nelle due direzioni *x* e *y*, questo sarebbe troppo bello! SCALE esegue automaticamente calcoli che sarebbero altrimenti necessari per convertire i valori che noi possiamo voler immettere, per esempio nel disegnare una curva.
La sintassi di SCALE è

SCALE 1, *x*, *y* oppure SCALE 0

Così:

SCALE 1 senza *x*, *y* equivale a SCALE 1, 1023, 1023
SCALE 0 torna alle coordinate di default (319, 199).

SCALE 1, 640, 200 consente di ottenere gli stessi risultati con una stessa riga di programma usando sia GRAPHIC 1/2 che GRAPHIC 3/4.
Notare che un comando GRAPHIC ha lo stesso effetto di SCALE 0: questo significa che se si cambia modo grafico durante il lavoro, è necessario ripetere il comando SCALE. Si veda il programma SCALE.

7.8 File di bit map

Si è già accennato nel Capitolo 3 all'uso di BSAVE e BLOAD per salvare non caratteri ASCII ma il contenuto della memoria in forma binaria. La System Guide lo propone soltanto per gli sprite. Il programma SALVAGRAFICI del dischetto consente di salvare invece tutto uno schermo grafico.
Il bit map completo occupa 37 blocchi di 255 byte sul dischetto, anche se per i disegni in modo GRAPHIC 1, BSAVE "*nomefile*", B0, P7186 TO P16208 è sufficiente e produce un file di 36 blocchi. Questo è comunque dispendioso se si lavora con i drive 1541 o 1570 (a un solo lato).
Inoltre, serve un altro file (il file "cf." sul nostro dischetto) di 5 blocchi salvato da un'area diversa di memoria (banco 15), se si desidera conservare le informazioni relative alla fonte 3: senza questo file tutte le zone colorate con la fonte 3 avranno lo stesso colore.
Quattro blocchi sul disco corrispondono a un kilobyte: infatti l'intero schermo grafico, come si è già detto, occupa 9 K. Ma dopo aver salvato il grafico con questo metodo si può eliminare dal programma tutte le istruzioni necessarie per creare il grafico. Sarà sufficiente usare BLOAD. Per capire il significato di questo, in termini di tempo e di complessità del programma, usare prima il programma ELLISSE sul Menu Comandi Grafici; poi usare ORA ESATTA. Nel secondo caso, sono necessari sei secondi per riprodurre sullo schermo il disegno dell'orologio che ne impiega oltre 60 se si crea ex novo con ELLISSE. Il file GR.OROLOGIO che dà questa possibilità è il solito file binario di tutto lo schermo, ottenuto interrompendo l'esecuzione di ELLISSE quando l'orologio era già sullo schermo, e salvando il contenuto della memoria grafica con BSAVE tramite SALVAGRAFICI. Il programma ORANDXOR esegue la stessa operazione sul file GR.LOGO, per fare dei giochi irriverenti con il marchio della McGraw-Hill usando gli operatori booleani.

8

I comandi grafici

Questo capitolo esamina i comandi CHAR, DRAW, BOX, CIRCLE e SCALE. Si è già parlato di CHAR con riferimento al suo uso sugli schermi di testo.

8.1 CHAR

CHAR è l'unico modo di inserire una dicitura in mezzo a un grafico senza creare caratteri speciali e, come sul normale schermo di testo, consente di determinare accuratamente la posizione delle parole. Mentre PRINT non funziona sullo schermo grafico (se non si apre una finestra sullo schermo normale a 40 colonne con GRAPHIC 2 o GRAPHIC 4), CHAR ha la capacità di scrivere dappertutto sul video (ma non sulla stampante, dove sarebbe molto utile).

```
10 color 0, 16: graphic 1, 1: color 0, 3
20 char 0, 5, 2, "questo e´ lo schermo grafico",1
30 color 1, 2
40 char 1, 5, 5, chr$(14) + "Questi Caratteri Sono Ottenuti con"
50 color 1, 8
60 char 1, 5, 7, "char",1
```

La sintassi è:

CHAR [*fonte colore*], *x*, *y*, *stringa* [,*rvs*]

Qui *x* va da 0 a 79 e *y* da 0 a 24. Poiché il numero dei caratteri sullo schermo grafico e su quello a 40 colonne è di 40 (0-39), i numeri superiori formano una seconda riga.
Un fatto non spiegato chiaramente nella System Guide è come ottenere caratteri maiuscoli e minuscoli sullo schermo grafico. CHAR, in modo GRAPHIC, dà solo maiuscoli e caratteri grafici: per avere maiuscolo e minuscolo è necessario aggiungere CHR$(14):

```
CHAR1,5,15,CHR$(14)+"Stringa MAIUSCOLO e Minuscolo"
```

Nelle righe elencate sopra la fonte del colore è sempre 0 o 1, cioè il risultato delle specifiche COLOR 0 e COLOR 1. Usando GRAPHIC 3, *fonte colore* può anche avere il valore 2 o 3. Il programma pulisce lo schermo grafico con GRAPHIC 1,1 facendogli quindi assumere il colore 16 (grigio chiaro) già in vigore (dopo COLOR 0, 16) per lo schermo di testo. La riga 20 fa stampare le parole QUESTO È LO SCHERMO GRAFICO in rosso nella colonna 5 della riga 3.
Queste parole appariranno in maiuscolo: è quindi importante evitare di usare il tasto SHIFT, perché questo farebbe apparire caratteri grafici e non le maiuscole, se non si usa CHR$(14), come nella riga 40.
La riga 60 fa apparire CHAR in campo inverso (cioè in negativo), grazie a ,1 dopo la stringa. La stringa può essere una variabile (come A$) o anche una combinazione del tipo

```
CHAR 1, 3, 10, a$+"fine"
```

Le coordinate di CHAR sullo schermo grafico sono sempre quelle dello schermo a 40 colonne e non sono influenzate da SCALE.
CHAR, come si è accennato altrove, per esempio nel paragrafo 9.5 (WINDOW), è molto utile anche sullo schermo normale a 40 o a 80 colonne. Sullo schermo normale si possono usare gli speciali caratteri del colore, ottenuti premendo CONTROL o COMMODORE insieme a un numero: questo sullo schermo grafico non ha effetto.

8.2 DRAW

L'uso di questo comando è fondamentale, e costituisce la base della serie di tre programmi DRAW, JOYDRAW e MULTIDRAW sul Lato 2 del dischetto. Si rimanda a questi per un uso pratico che fornirà il necessario addestramento. Sul Lato 1 del disco il tutorial DRAW da un'illustrazione sintetica.
DRAW non è complesso:

DRAW *fonte colore, x, y* disegna un punto.
DRAW *fonte colore, x1, y1* TO *x2, y2* [TO *x3, y3...*] disegna da un paio di coordinate al successivo
DRAW TO *xn, yn* TO *xnn, ynn* TO... disegna dall'ultimo punto al successivo.

Con l'ultima espressione, la riga parte dall'ultima posizione del pixel cursor. Se non si sa dov'era si possono ottenere i valori con:

x =RDOT(0)
y =RDOT(1)
fc=RDOT(2)

Naturalmente, in modo diretto la funzione RDOT può essere usata con PRINT RDOT(*n*).
Se per qualche motivo non si vuole usare DRAW in una delle prime due forme elencate sopra, ma si preferisce la terza, si può usare l'istruzione LOCATE nella forma:

LOCATE *x, y*

Questo metterà il cursore alla nuova posizione senza tracciare una riga.
Vale la pena di annotare qui che DRAW, come vari altri comandi grafici, accetta i parametri anche in un modo non descritto dalla System Guide, probabilmente perché danno risultati difficili da descrivere e perché non offrono tutte le possibilità che ci potremmo aspettare.
Si vedano la voce MOVSPR nel Dizionario e il Capitolo 12 per un'esposizione delle coordinate relative con e senza angolo. Il principio è illustrato in una sintassi del tipo

DRAW TO +50+50
DRAW TO +50; 90

Il primo disegna una riga dalla posizione corrente *x, y* del pixel cursor (leggibile con la funzione RDOT) fino a una posizione *x*+50, *y*+50. Il secondo disegna una riga da *x, y* lunga 50 e ad un angolo di 90 gradi.
Anche CIRCLE, BOX e PAINT accettano coordinate del genere. Non sembra possibile usare coordinate negative come con MOVSPR. Si consiglia (se interessa) di fare degli esperimenti per sondarne le possibilità.

8.3 BOX

Quest'istruzione disegna un rettangolo. È possibile anche ruotare il rettangolo e/o riempirlo di qualsiasi colore. Il tutorial BOX ne illustra alcune possibilità elementari.
La sintassi di BOX è, come dice il programma:

BOX [*fonte colore*], *x1, y1,* [,*x2, y2*] [,*angolo*] [,*paint*]

fonte colore può essere omessa (ma non la virgola dopo): lo stesso vale per *,angolo* e *,paint*. Se si omettono sia *x2* che *y2*, è necessaria una sola virgola. L'opzione *angolo* è in gradi. Il *,paint* è un valore 0 o 1 (0=NO e non è necessario). Con *,paint*=1, non viene disegnato separatamente il bordo: un bordo invisibile si riempie di colore.
Il programma disegna un rettangolo vuoto e orizzontale (riga 90): poi uno simile ma ruotato rispetto all'orizzontale (riga 190), poi un terzo, anch'esso obliquo, riempito di verde (righe 210-220). Infine un effetto "artistico": una sigla di fine programma fatta con un rettangolo lungo 200 pixel ma largo solo 2. Ne risulta una riga un po' irregolare che è colorata in bianco ma assume tinte diverse grazie a certe aberrazioni cromatiche sullo schermo. Il loop che comincia a 320 lo disegna e lo ridisegna variando il parametro, *angolo* a step di 3 gradi. Il lettore, se in vena di esperimenti, potrebbe variare le coordinate nella riga 330 e osservarne gli effetti.

8.4 CIRCLE

Quest'istruzione dovrebbe chiamarsi ELLIPSE, perché disegna (tra l'altro) questa forma geometrica e non solo il cerchio, che è un caso particolare dell'ellisse. Nei tre programmi DRAW, JOYDRAW e MULTIDRAW il comando è stato chiamato E (ellisse).
La sintassi di CIRCLE è più complessa: affrontiamo prima le coordinate *x,y* e i raggi.

CIRCLE *fonte colore, x, y, xr, yr*

Qui *fonte colore* è la già nota fonte del colore, mentre *x* e *y* sono le coordinate del centro del cerchio. Poi si definiscono i raggi *xr, yr*. Un cerchio ha un solo raggio, e quindi si definisce

CIRCLE 1, 150, 100, 50, 50

Ma il BASIC consente di omettere il secondo raggio se non è diverso dal raggio *x*. Si può scrivere quindi

CIRCLE 1, 150, 100, 50

Se però avessimo bisogno di mettere altre indicazioni dopo il raggio *y* (vedremo quali più avanti), bisognerebbe inserire la virgola corrispondente, anche se non si inserisse un valore.
Il seguente programma disegna tre ellissi, una delle quali è un cerchio:

```
Ellisse.list
10 width2: color0,16: color1,4: gosub220
20 graphic1,1
30 circle , 159, 100, 100
40 color 4, 1: color 1, 4
60 paint 1, 0, 0: color 1, 15
70 paint , 319, 0
80 circle , 159, 100, 90, 50
90 circle , 159, 100, 80, 25
100 color 1, 4
110 paint 1, 100, 65
120 color 1, 7: paint , 150, 25
130 paint , 150, 85
140 color 1, 1: char , 15, 4, "ora esatta", 1
160 do
170 : char , 17, 10, "ore:" + mid$(ti$,1,2)
180 : char , 17, 12, "min:" + mid$(ti$,3,2)
190 : char , 17, 14, "sec:" + mid$(ti$,5,2)
200 loop
210 end
220 print "Sqqqq  Inserire l'ora esatta (hhmmss)
230 input  ti$
240 return
ready
```

Chiede l'ora esatta e poi la presenta, come orologio digitale, su uno schermo grafico. Inserisce l'ora nei sei byte di TIME$ e in seguito usa MID$ per leggere le ore, i minuti e i secondi, illustrando così una delle funzioni più utili di CHAR (nonché di MID$). Si osservi il modo di concatenare insieme i vari pezzi di stringa (per esempio nella 170).
Dopo la subroutine, il programma disegna prima un grosso cerchio (riga 30), poi dipinge di cyan la parte dello schermo a sinistra del cerchio, e di blu chiaro la parte destra.
Le righe 80 e 90 danno invece due ellissi concentriche. Ora avviene una serie di PAINT che sfruttano ciò che è in realtà un difetto della risoluzione del colore nel modo GRAPHIC 1/2, eliminando tutte le linee curve per produrre un effetto a blocchi. Come abbiamo già osservato, i vari byte si riempiono

interamente di un colore o di un altro, e quindi i bordi prodotti da CIRCLE diventano una serie di gradini. Non sono esattamente simmetrici: un perfezionista dovrebbe sperimentare con le coordinate centrali delle ellissi fino a trovare quelle capaci di centrare esattamente i byte.
L'utilità di questo programma è limitata per il tempo richiesto nel disegno. Ma per usare uno schermo del genere non è necessario usare il programma per disegnarlo. Infatti, il programma ORA ESATTA usa un file binario creato con SALVAGRAFICI, per ottenere lo stesso risultato in 4-5 secondi.

```
ora esatta.list
10 color 0, 16: color 1, 4: gosub220
20 graphicl, 1
140 bload "orologio", b0, p7168
160 do
170 : char , 17, 10, "ore:" + mid$(ti$,1,2)
180 : char , 17, 12, "min:" + mid$(ti$,3,2)
190 : char , 17, 14, "sec:" + mid$(ti$,5,2)
200 loop
210 end
220 print "  Inserire l´ora esatta (hhmmss)
230 input  ti$
240 return
```

In questa forma il tempo richiesto è così limitato che la routine potrebbe essere impiegata anche più volte all'interno di uno stesso programma, e, se non si usano altri schermi grafici, non sarà necessario caricare il disegno che una sola volta.
Con CIRCLE si possono produrre anche archi (a partire da un angolo qualsiasi fino a un altro angolo qualsiasi) e, per un complesso gioco matematico, è possibile creare altre forme (ottagono, pentagono, endecagono, trapezio, rombo, triangolo), e anche farle ruotare. Le opzioni addizionali sono:

CIRCLE [*fonte colore*], *x,y,xr* [*,yr*] [*,iniz*] [*,fine*] [*,rot*] [*,inc*]

dove *,iniz* indica l'angolo dove deve iniziare il disegno (0=mezzogiorno e si procede in senso orario) e *,fine* indica l'angolo al quale termina.
L'opzione *,rot* indica l'angolo di rotazione in senso orario (default 0), mentre *,inc* è l'angolo incluso, cioè i gradi tra segmenti (default 0).

```
20 draw , 150, 100 to 150, 0: rem raggio
30 circle , 150, 100, 100, , , 45
50 draw to 150, 100 : rem raggio
60 getkey a$: graphic 1,1
```

```
70 circle , 150, 100, 50, 50, 90, 270:
   rem: arco con iniz. e fine
90 getkey a$: graphicl,l
```

Le righe 20-50 disegnano un arco di 45 gradi (*iniz*=0, *fine*=45) con due raggi che lo congiungono al centro. GETKEY nella 60 ferma il programma per osservarlo.
La 70, invece, disegna un semicerchio (cioè un arco di 180 gradi) a forma di coppa aperto verso l'alto: va infatti da 90 (*iniz*) a 270 gradi (*fine*).

```
110 circle , 150, 100, 100, , , , , 360/11: rem: endecagono
130 getkey a$: graphicl,l
```

Qui, omettendo *yr, iniz, fine, rot,* specifichiamo un incluso di 360/11 gradi, che produce un poligono regolare di 11 lati. Le righe seguenti danno altri esempi, e il programma INC (disponibile solo dal Menu Finale di CIRCLE) consente di provare tutti i valori.

```
150 circle , 150, 100, 100, , , , , 45: rem: ottagono
190 circle , 150, 100, 100, , , , , 72.5: rem: pentagono
230 circle , 150, 100, 100, , , , , 90: rem: rombo
270 circle , 150, 100, 100, , , , , 120: rem: triangolo
310 circle , 150, 100, 100, , , , 22, 105:'rem: trapezio
```

Se nella 310 si elimina la *rot* di 22 gradi, il trapezio appare storto. Per meglio illustrare gli effetti di *rot*, è stato costruito il piccolo loop che segue. La variabile t aumenta la rotazione a step di 15 gradi, producendo quindi 12 triangoli sovrapposti. Ecco un'altra possibilità da usare nella sigla d'inizio di qualche programma con pretese d'eleganza.

```
350 rem: 12 triangoli con rotazione
360 for t = 0 to 180 step15
370 :  circle , 150, 100, 100, , , , t, 390
380 next
400 getkey a$: graphic0: color 0,16
```

I programmi grafici descritti nel tutorial alla fine di questo capitolo permetteranno al lettore di provare queste opzioni personalmente.

8.5 SCALE

Ritorniamo un momento al comando SCALE. Ecco un programma brevissimo (come numero di righe ma non come tempo di svolgimento), che ne illustra il funzionamento:

```
scale.list
 10 trap 170
 20 width 2: color 0, 1: color 1, 2: color 4, 1: color 5, 1
 40 graphic 1, 1: color 0, 4
 60 for t = 350 to 2000 step 10
 70 :  scale 1, t, t
 80 :  circle , 280, 260, 30, 70
 90 :  char , 19, 24, "scale 1," + str$(t) + "," + str$(t) + "     "
100 :  color 1, 4: box 1, 340, 70, 260, 130, , 0: color 1, 15
110 :  circle 1, 80, 290, 50, 90, , , , 120
120 :  color 1, 2
130 :  draw 1, 160, 300 to 160, 220: color 1, 2
150 :  draw 1, t-1, t-1
160 next
170 getkey a$: color 0, 16: color 1, 1: graphic 0: color 5, 1:
    list: end
```

Questo programma abolisce la scala normale ($x=0-319$, $y=0-199$), e parte con $x=0-350$, $y=0-350$ (riga 70). Il loop incrementa la scala a step di 10, fino a raggiungere un massimo di 2000, 2000.
La riga 150 disegna un punto (DRAW con solo *x*, *y*) nell'angolo inferiore destro, corrispondente alla scala massima per *x* e per *y*. Poiché questo punto è di un solo pixel, viene fatto lampeggiare grazie al " " alla fine della riga 90, che usa STR$ per presentare sullo schermo le coordinate in vigore.
La 80 disegna un'ellisse in basso a destra, la 100 un rettangolo più in alto a destra, la 110 un triangolo in basso a sinistra, e infine il DRAW della 130 disegna una piccola riga verticale. Queste operazioni sono tutte all'interno del loop che costituisce quasi l'intero programma, e si ripetono quindi molte volte prima che si arrivi alla scala di 2000, 2000.
Che cosa accade? A ogni giro del loop, la scala diventa più grande mentre le coordinate nelle varie specifiche restano tali e quali. Ne risulta che, a ogni giro, le diverse forme geometriche vengono disegnate un po' più piccole, un po' più in alto e un po' più a sinistra.
I valori massimi di SCALE sono oltre 65000, ma è chiaro che non cambia niente.
Per il TRAP all'inizio, anche la pressione del tasto STOP conta come errore. Basta premere prima STOP e poi RETURN per fermare il programma in qualsiasi istante, il che può essere utile data la sua durata.

8.6 Grafica sul disco

Abbiamo visto quante possibilità per la produzione di disegni sono offerte dal BASIC 7.0: ciascuno dei comandi grafici è abbastanza semplice, ma il numero delle scelte possibili, e le infinite combinazioni che ne possono

derivare, significano che il programmatore deve avere dell'immaginazione. Finché non ha visto sullo schermo gli effetti ottenibili, succederà che molti tentativi produrranno risultati diversi dal previsto.
In particolare questa osservazione è importante per quanto riguarda la scala e la fonte del colore. Per la scala, si tratta semplicemente di acquisire la capacità di stimare, per esempio, quanto dev'essere grande un cerchio o un rettangolo. La fonte o sorgente del colore, invece — argomento toccato in modo molto teorico fino a questo punto — richiede una pratica seria prima che si riesca a sfruttare le possibilità.
Il Lato 2 del disco costituisce in pratica un pacchetto grafico piuttosto completo. Supponendo che il lettore abbia già un'idea almeno teorica dell'uso dei vari comandi, ottenuta sia dalle pagine precedenti che dai programmi GRAFTUT, CIRCLE, BOX, SCALE, ecc. (sul Lato 1 del disco), si propone qui un tutòrial creativo... un gioco. Si tratta semplicemente di disegnare o anche di pasticciare usando ciascuno dei tre programmi, fino a essere in grado di prevedere il risultato delle operazioni e di sfruttare tutte le notevoli possibilità disponibili.
I tre programmi consentono di eseguire disegni al tratto, formare cerchi (ellissi), rettangoli, dipingere con PAINT: insomma di mettere in pratica tutto ciò che abbiamo finora esaminato. Quanto segue è un piccolo manuale d'uso del sistema grafico: il tutorial vero e proprio consiste nell'uso libero dei programmi. Le righe di programma commentate di seguito provengono in genere da MULTIDRAW; sono comunque più o meno identiche negli altri due programmi.
Le osservazioni su DRAW valgono, se non è specificato diversamente, per gli altri due programmi.

DRAW

Questo è il programma più semplice. Usa soltanto le fonti 0 (sfondo) e 1 (colore principale). Il colore dello sfondo viene stabilito prima di cominciare il disegno, ma in seguito si può cambiare il colore nella fonte 0 e usarlo nel disegno. Il modo grafico è GRAPHIC 1/2. DRAW presenta tutte le opzioni fondamentali, ma non l'uso di GSHAPE per copiare piccole aree di disegno e altre possibilità offerte dal programma JOYDRAW.
Nonostante la sua semplicità, è un programma utile, per i disegni al tratto e in particolare per disegnare zone piccole, come caratteri speciali. Dopo aver eseguito un lavoro del genere, si può passare a uno degli altri due programmi senza perdere il disegno.
Come negli altri due programmi, esiste un loop principale nel quale si possono disegnare solo linee (DRAW). In questo programma si usano i tasti-freccia per spostarsi senza disegnare e i tasti della tastiera numerica [8=SU, 2=GIÙ, 4=SINISTRA, 6=DESTRA, 1,7,3,9=diagonali]: ogni

pressione di uno di questi tasti corrisponde a uno spostamento di un solo pixel (punto).
Dal loop principale, che in tutti i programmi comincia alla riga 500, si può uscire soltanto premendo STOP. L'istruzione TRAP 1000 all'interno di questo loop fa sì che, invece di terminare, il programma vada ad affrontare la serie di loop che inizia appunto a 1000.
La possibilità di usare STOP viene sospesa con la riga:

```
1000 trap500: poke808,100: window0,22,39,24
```

e quando si torna al loop principale si riattiva con:

```
500 do: print"S": poke808,110
```

(Si veda la voce POKE nel Dizionario per una spiegazione di queste e altre operazioni). A questo punto si passa a GRAPHIC 2 e la finestra presenta un piccolo menu:

P=Paint	F=Fonte	!=Fine	S=Salti	B=Box	E=Ellisse
W=Larghezza		H=Aiuto	L=Linea	T=Tratto	

L'opzione FONTE consente di cambiare il colore in una fonte (COLOR F,C) mentre T consente di selezionare la fonte con la quale si disegna usando le frecce (o il joystick).
Premendo H si ha una serie di schermate di aiuto, tranne che per DRAW, che essendo più semplice riesce a usare una sola schermata. Premendo W (WIDTH) si accede alla breve routine:

```
1240 ifa$ = "w" and ww = 0 then width2: ww = 1: goto500:
     else ifa$ = "w" then width1: ww = 0: goto500
```

In sintesi: se il tratto è largo, premendo W si avrà un tratto stretto, e viceversa. Se invece a$="b":

```
1300 ifa$ = "b" then begin
1310 :   xc = 0:yc = 0:p = 0:an = 0
1320 :   print "coord. alto a sin. x = "; x; " y = "; y
1330 :   print "Basso a destra.    : x = x + "; : input xc
1340 :   print "                   : y = y + "; : input yc
1350 :   input "angolo"; an
1360 :   input "paint (s/n)"; q$: ifq$ = "" then 1360
1370 :   input "Fonte"; f
1380 :   if q$  = "s" then p = 1
1390 :   boxf, x, y, x + xc, y + yc, an, p: goto500
1400 bend
```

Il lettore vedrà che si possono specificare tutti i parametri di BOX. Analogamente alla riga 1500 inizia il loop per a$="e". La "e" per ellisse serve a ricordare che CIRCLE non fa soltanto cerchi, ma anche ellissi e (con l'opzione *inc*) anche triangoli, ottagoni e rombi come spiegato in precedenza.

```
1500 ifa$ = "e" then begin
1503 :   al = 0: a2 = 360: a3 = 0: a4 = 2
1505 :   print "Opzioni? (s/n)": getkey a$: ifa$ <> "s" then 1560
1510 :   input "angolo iniziale arco"; al
1520 :   input "angolo finale arco"; a2
1530 :   input "gradi rotazione"; a3
1540 :   input "angolo incluso (poligoni)"; a4
1560 :   rx = 0: ry = 0
1565 :   input "raggio x"; rx
1570 :   input  "raggio y"; ry: ifry = 0 then ry = rx
1575 :   input "Fonte"; f
1580 :   circle f, x, y, rx, ry, al, a2, a3, a4
1585 bend: goto500
```

Il loop per PAINT inizia a 1700, mentre a 1800 inizia il loop per a$="S" e per a$="L". "S" sta per "saltare". Nei due programmi successivi questo loop assume un'importanza fondamentale: qui consente di saltare gruppi di 8 pixel per volta, semplicemente per rendere più veloce il movimento: con le frecce ci si muove sempre di un pixel. Se invece si è selezionato "L", si entra nello stesso loop, con la differenza che, ogni volta che si preme L, verrà disegnata una linea dall'ultimo punto in cui si trovava il cursore fino a quello attuale:

```
2370 :    if v$ = "l" and a$ = "l" then draw ft, xc, yc to x, y:
          xc = x: yc = y: gotol910
```

Questo rende più facile disegnare righe lunghe, in particolare quelle diagonali.

Il programma termina con una routine d'uscita che consente di salvare il grafico con SALVAGRAFICI, di ritornare al menu principale, di iniziare un nuovo disegno o di usare TAGLIACUCI (si veda più avanti). È importante concludere sempre il lavoro ricaricando il menu principale: poiché si usano dei POKE nella locazione 4184 per tenere informato il calcolatore circa la situazione del lavoro (mentre si caricano e si ricaricano i vari programmi), solo il menu consente di riportare 4184 al suo valore normale (decimale 77). La cosa ovviamente non ha importanza se si intende spegnere il calcolatore. Se (ma non dovrebbe capitare) ci si trovasse fuori da qualsiasi programma, con in memoria un disegno importante, eseguire POKE 4184,128, e poi RUN *nomeprogramma*, dove *nomeprogramma* è

uno dei 3 programmi di disegno, SALVAGRAFICI o TAGLIACUCI. Se il valore in 4184 è 128, il programma caricato salterà i menu iniziali, non pulirà lo schermo grafico e ci si troverà pronti a riprendere il lavoro.

JOYDRAW

Questo programma, come MULTIDRAW, aggiunge numerose altre possibilità a DRAW.
In primo luogo, sullo schermo di disegno appare una tavolozza che utilizza 3 sprite per far vedere i colori attualmente inseriti nelle due fonti disponibili. Questi contengono i numeri 0 e 1, mentre il terzo sprite è una T, ed indica il colore attualmente "sotto il cursore". Per cambiare il colore sotto il cursore, esistono due possibilità:

- Premere STOP e poi T. Questo serve per cambiare la fonte del tratto: il calcolatore chiederà "Fonte?".
- Premere STOP e poi F. Questo serve per cambiare il colore contenuto in una fonte: inserire il numero della fonte che si desidera cambiare: il calcolatore chiederà poi "Colore?". Inserire un numero da 1 a 16 e premere RETURN.

A questo punto conviene fare delle prove, eseguendo dei PAINT e disegnando righe con vari colori tratti da fonti diverse. HELP è disponibile anche mentre si usa il joystick: basta premere H oppure HELP.
Una nota utile: se non si vede lo sprite a forma di freccia che costituisce il "pennello", premere B o N per cambiarne il colore in bianco o in nero. Se (sullo schermo dei salti) si ha invece il cursore a telaio, si può tornare alla freccia premendo F.
Notare che, in modo GRAPHIC 1/2, una volta stabilito il colore dello sfondo (fonte 0), questo resta: se poi si cambia il colore contenuto nella fonte 0, si può usare questo altro colore. Se il tratto viene dalla fonte 1, si avranno righe nitide: se il tratto viene dalla stessa fonte 0, si avranno i colori che "spandono" (ma questo può anche essere utile, producendo un disegno a tratti grossi, quasi "naif").
Ora caricare MULTIDRAW, che ha invece 4 fonti (0-3) a disposizione, e fare altri esperimenti analoghi.
Una grande differenza: in modo GRAPHIC 3/4, se cambiamo la fonte 0, cambia anche il colore già presente sullo schermo. Questo è utilissimo per preparare grafici che possono essere utilizzati da altri programmi per dare un effetto di lampeggiamento. Provare:

```
1 for t=1 to 16: graphic 3:color 0,t: sleep 1:next: goto 1
```

MULTIDRAW è una copia quasi esatta di JOYDRAW, modificata soltanto

nel senso che specifica GRAPHIC 3 e GRAPHIC 4 anziché 1 e 2. Sia JOYDRAW che MULTIDRAW offrono ben altre possibilità:

- Duplicazione/cancellazione di una zona del disegno. Premere STOP e S: sul Menu dei Salti scegliere g (minuscola). Il cursore diventerà un telaio o "mirino" grande quanto uno sprite. Per duplicare una zona, usare i soliti tasti per portare il telaio esattamente sopra la zona desiderata e premere il tasto ^ (la freccia verticale accanto al tasto RESTORE). Questo comando usa SSHAPE e GSHAPE per copiare la zona:

```
2150 :    if v$ = "" then sshape s$, x-20, y-20, x + 2, y:
          sprsav s$, 1: a$ = "gl"
```

Adesso possiamo spostare il cursore-telaio nel solito modo. Trovata la zona giusta, basta premere r (minuscolo) per stampare la zona in quel punto:

```
2220 :    ifv$ = "r" then gshape s$, x-20, y-21:
          sprsavs$, 1: a$ = "gl": goto1910
```

Ma se invece preferiamo vedere il telaio, possiamo cambiare il cursore nella zona stessa premendo m (minuscola): ora vedremo spostarsi il pezzo di grafico: r serve sempre per stamparlo. Ecco un modo di produrre disegni ripetitivi accuratamente. Premere F per riottenere un cursore normale.

- Per cancellare una zona, basta copiare una zona di sfondo vuota, poi portare il telaio sopra la zona da eliminare. Per fare ciò, è opportuno duplicare una zona di fonte 0 colorata con il colore (il default è 16) dello sfondo iniziale. Per eliminare una zona grande usare BOX con "Paint?"=sì e "Fonte?"=0, oppure (con le dovute cautele) PAINT con Modo 0.
  Il colore e la fonte con i quali si riproduce la zona possono essere cambiati: basta premere q per passare nel loop principale, modificare colori e fonti, e tornare ai Salti: ovviamente la zona in memoria resta finché non la modifichiamo.
- Inserimento di caratteri e duplicazione di piccole zone. Sul Menu dei Salti il comando k consente di usare CHAR per inserire sullo schermo grafico uno o più caratteri:

```
2100 :    if v$ = "k" or v$ = "K" then begin: u = 2:
              sprsavk2$, 1
2110 :      input "Caratteri"; c$
2115 :      input "Riga (0-24)"; nr:
              ifnr > 39 then 2115
2117 :      input "Colonna (0-39)"; nc:
```

```
                    if nc > 39 then 2117
2120 :          input "Fonte"; ft
2130 :          char ft, nc, nr, chr$(14) + c$
2140 :      bend: goto1860
```

Anche se si possono specificare le coordinate (x=0-39, y=0-24), la procedura più comoda è di inserire la stringa in alto a sinistra (riga 2 o 3) e poi spostarla: in seguito si ripulisce la zona in alto a sinistra. Per spostarla esistono tre possibilità: la prima è quella già descritta, con g, poi ^ e infine r.

- Premendo invece G (Maiuscola) o T (Maiuscola) si presenta un altro cursore a telaio più piccolo, grande esattamente quanto una W maiuscola. Centrando esattamente un carattere si hanno le stesse possibilità: premendo * si salva, in un'altra stringa (k$), il carattere nel telaietto. Ora si procede come prima, usando però le maiuscole: R per stampare, M per avere il cursore uguale alla zona (in questo caso il cursore può sembrare più grande, ma la zona che verrà stampata sarà quella voluta). La finestra in fondo allo schermo indica quali lettere usare. Questa possibilità serve per inserire caratteri in qualsiasi punto dello schermo: CHAR, infatti, non funziona con SCALE, e posiziona sempre i caratteri in colonne (0-39) e righe (0-24). Noi possiamo posizionare un carattere su qualsiasi pixel dello schermo. È quindi possibile anche stringerli insieme e risparmiare un po' di spazio.

MULTIDRAW

MULTIDRAW funziona esattamente come JOYDRAW (e come abbiamo detto le varie routine principali cominciano agli stessi numeri di riga). La tavolozza è più estesa, con 4 colori anziché 2, e richiede quindi un po' più di attenzione, mentre in compenso offre una più larga gamma di possibilità, anche se la risoluzione dell'immagine in senso orizzontale è dimezzata. I caratteri ottenuti con CHAR sono molto meno belli.
Questo è il programma da usare per abituarsi ad alternare le fonti, tenendo sempre presente che una riga disegnata con fonte 3, per esempio, che attraversa un'altra riga di quella fonte o una zona dipinta con quella fonte, non sarà visibile se il colore nella fonte è sempre lo stesso; mentre, se si cambia il colore, questo "spanderà". Basta invece inserire il colore voluto in un'altra fonte e usare quest'ultima per disegnare.
Un esperimento utile per capire rapidamente è usare PAINT per dipingere tutto lo schermo con una fonte diversa da 0. Poi eseguire disegni, BOX, ecc., con tutte le altre fonti, prestando particolare attenzione a ciò che accade con fonte 0 e con i vari colori in questa fonte.
Se tutto ciò sembra complesso, è perché si ha la possibilità di agire in numerosi modi su numerosi parametri: non può quindi essere semplice.

Ma una volta che il lettore avrà acquisito una certa dimestichezza con i comandi, capirà anche che il BASIC 7.0 non poteva rendere più facile il suo lavoro.

TAGLIACUCI e MULTICUCI

Questi due programmi usano SSHAPE e GSHAPE come JOYDRAW e MULTIDRAW, ma invece di riprodurre aree piuttosto piccole, possono riprodurre aree di tutte le dimensioni: dallo schermo intero fino a una zona piccolissima.
Permettono quindi di esercitarsi con l'uso di SSHAPE e GSHAPE, particolarmente per quanto riguarda il parametro *modo*. Si veda prima di usarli il programma SHAPES sul Lato 1 del disco.
Entrambi i programmi consentono di:

- Caricare un grafico bit-map, ritagliarne una parte e riprodurla in un altro punto dello stesso grafico, anche variando il Modo. Per esempio, se *modo*=3 si avrà la sovrapposizione della zona riprodotta; con *modo*=1 la riproduzione in negativo.
- Eseguire la stessa operazione di ritaglio, poi caricare un secondo grafico e riprodurre la zona ritagliata dal primo.

Sono abbastanza flessibili nell'uso, e possono dare risultati sorprendenti. Oltre alle riproduzioni ovvie, provate a spostare i cursori, dopo aver duplicato una zona, di 1 o 2 pixel, e ripetere la duplicazione. Specie in Modo 4, si arriva a produrre disegni strani e piuttosto belli.
Notate che entrambi usano il modo FAST, per cui lo schermo grafico diventa vuoto mentre SSHAPE salva la zona ritagliata. È sufficiente attendere senza premere tasti.

9

40 e 80 colonne: gli schermi del testo e le finestre

Il C-128 è dotato di 16 K di memoria addizionale dedicati allo schermo a 80 colonne. Per motivi meglio noti alla Commodore, questa memoria non è accessibile tramite il BASIC. In particolare i comandi grafici non possono essere usati. Immediatamente dopo la comparsa sul mercato del C-128, questo fatto ha rappresentato una sfida irresistibile e molti si sono messi a cercare modi di aggirare l'ostacolo. Sono già stati pubblicati numerosi programmi di grafica in "ultra high resolution", cioè con una risoluzione di 640×200 pixel. Non risulta finora che qualcuno abbia trovato il modo di ottenere disegni con più di due colori, né che esistano programmi per produrre gli sprite su questo schermo.
Sarà opportuno però partire da una considerazione sul meccanismo con il quale si riproducono i caratteri sullo schermo. Abbiamo già visto, nel Capitolo 7, che esistono 3 tipi di codici per i caratteri:

- codice ASCII standard
- codice ASCII-CBM (aggiunge i caratteri grafici)
- codici di schermo.

Il codice di schermo (si veda la tabella nell'Appendice D della System Guide) è il valore che, per lo schermo di testo a 40 colonne, si trova in ciascuno degli indirizzi tra 1024 e 2023 del banco 0, e che indica al calcolatore quale gruppo di 8 byte deve andare a cercare nella ROM e riprodurre sullo schermo in ciascuna delle 1000 posizioni disponibili sullo schermo. In questo codice il valore 1 corrisponde alla lettera a (minuscola), e il valore massimo è 127. In realtà i caratteri disponibili sullo schermo sono 256, poiché PRINT CHR$(14) e PRINT CHR$(142), oppure la

pressione simultanea dei tasti COMMODORE e SHIFT, modificano i puntatori (cioè i valori) che indicano quale zona della ROM-caratteri deve essere presa in considerazione. Sullo schermo a 40 colonne si possono avere gli uni o gli altri: mai entrambe le serie insieme, mentre in modo 80 colonne l'informazione relativa a ciascun carattere viene registrata separatamente (basta un solo bit). Perciò il modo 80 colonne consente di visualizzare fino a 512 caratteri.
Le informazioni richieste per visualizzare un carattere non finiscono però a questo punto. Se dopo aver ripulito lo schermo a 40 colonne portiamo il cursore qualche riga verso il basso e inseriamo:

POKE 1024, 65

può darsi che non accada niente (apparentemente), se il carattere ha lo stesso colore dello schermo. Se premiamo HOME (non CLR), il cursore entra nello stesso quadretto dello schermo, e vediamo la nostra A, grazie all'effetto del colore del cursore. Il carattere è arrivato, ma non possiede un colore.
È necessario un secondo POKE per vedere la A in modo stabile: altri 1000 indirizzi vengono usati per salvare i dati relativi al colore, e questi vanno da 55296 a 56295.

POKE 55296,0

colorerà quindi la lettera di nero. Il valore di questo POKE deve essere sempre uno in meno del valore usato sulla tastiera con CONTROL: 0=nero, 1=bianco, 2=rosso, e così via.
Il programma PEEK/POKE legge una parte dello schermo di testo a 40 colonne, e ne copia il contenuto in una zona di RAM inutilizzata. È facile incorporare la routine di trasferimento in un programma proprio. Può essere utile per esempio per conservare il contenuto di una finestra (se si tratta di materiale inserito dall'utente) in modo da poter utilizzare lo schermo per altri scopi e poi ripristinarlo. Il sistema è lento, anche usando FAST: se si vuole salvare l'intero schermo, comporta infatti 2000 operazioni di PEEK e 2000 POKE, sia per copiare lo schermo sia per ricopiarlo nelle posizioni originali. Nonostante ciò, il principio illustrato in questo programma potrà servire almeno in un caso.
Supponiamo di aver bisogno, in mezzo ad uno schermo pieno di testo, di aprire una finestra, di chiedere per esempio un dato all'utente, e poi di chiudere la finestra, restaurando il testo che prima era contenuto in quello spazio. Se si tratta di una finestra piccola, il metodo seguito nel programma sarà molto utile. Per salvare anche provvisoriamente l'intero schermo può essere più economico usare invece la routine che serve da esempio nel programma BSAVE, anch'esso sul Menu Comandi Vari. In

questo caso lo schermo viene salvato su disco e poi riletto con BLOAD: nulla vieta inoltre di usare SCRATCH alla fine del programma per eliminare il file provvisorio.
Lo schermo a 80 colonne è nuovo nell'ambito dei piccoli computer Commodore, anche se per il C-64 esistono sul mercato programmi (in particolare word processor) che offrono questa possibilità anche sul televisore domestico, in genere soltanto in bianco e nero. È invece uno standard sui personal (il PC offre anche 40 colonne) e sui calcolatori più grandi (i mini e i cosiddetti mainframe). Molti sistemi operativi e molti programmi professionali lo richiedono. Il CP/M del C-128 può funzionare con 40 colonne, ma è estremamente scomodo.

9.1 Il monitor a 80 colonne

Per ottenere i risultati migliori è necessario un monitor PAL/RGBI (RGBI - Red Green Blue Intensity). Il sistema PAL consente di visualizzare lo schermo a 40 colonne, che resta indispensabile per la grafica. I quattro segnali separati del sistema RGBI consentono una migliore definizione dell'immagine rispetto al segnale PAL. La risoluzione orizzontale (x) arriva a 640 pixel anziché 320, mentre quella verticale (y) è sempre di 200. Per motivi economici, molti sceglieranno un monitor diverso dal modello 1901 della Commodore. Esistono inoltre cavi che consentono di usare un monitor PAL (o un televisore usato come monitor con la presa SCART) per ottenere le 80 colonne. Questi ovviamente rappresentano la soluzione più economica e hanno il difetto di funzionare soltanto in bianco e nero. Appare invece impossibile ottenere un'immagine ragionevole su un televisore normale collegato con il cavo RF (antenna).
Con un monitor RGBI, anche se non è il 1901 della CBM, il risultato è più gradevole di quanto si ottiene con molti PC più costosi. Questi, infatti, spesso presentano caratteri lievemente più grandi, ma dall'aspetto più "granuloso": inoltre quando le righe "scrollano" sullo schermo, si ha un effetto di movimento a scatti che può dare fastidio agli occhi.
Usando un buon word processor si ha il vantaggio di visualizzare sullo schermo il testo esattamente come verrà stampato, e mentre si scrive, questo schermo consente di vedere una parte del testo grande circa quanto una normale pagina dattiloscritta.
È possibile usare qualsiasi colore di sfondo e di carattere (si veda la voce COLOR nel Dizionario del BASIC): quando si accende il calcolatore in modo 80 lo schermo è nero e i caratteri cyan. Cambiare i caratteri in verde (con CTRL oppure con COLOR 5,6) dà uno schermo indistinguibile da quello più comunemente usato sui terminali di sistemi multiutente come per esempio i sistemi UNIX.

Non è però molto chiaro per quale motivo si dovrebbe preferire il verde su sfondo nero. Sia per programmare che per scrivere testi, l'autore usa in genere uno sfondo bianco (COLOR 6,2) con carattere nero (COLOR 5,1).

9.2 I comandi per entrare in modo 80 colonne

ESC-X oppure PRINT CHR$(27);"X" — questo è on/off, cioè si ripete per tornare allo schermo precedente.
GRAPHIC 5 — GRAPHIC 0 riporta a 40 colonne
40/80 DISPLAY — il sistema parte in modo 80 se questo tasto è abbassato quando:

- si accende il calcolatore
- si premono STOP e RESTORE insieme
- si preme RESET.

La memoria per lo schermo a 80 colonne è indipendente da quello a 40: perciò, quando si è in modo 80, qualsiasi cosa già presente sullo schermo a 40 o su quello grafico rimane inalterato mentre si lavora in modo 80. È quindi possibile commutare fra i due schermi anche se purtroppo ciò richiede la pressione manuale di un commutatore sul monitor, e non è possibile effettuare questa operazione da tastiera o da programma. Se si ha però un altro monitor (o il TV di casa), si può usare quest'ultimo per lo schermo a 40 colonne, in modo simultaneo. Nei modi GRAPHIC 1 e GRAPHIC 3 infatti si avrà uno schermo grafico mentre l'altro (se è un monitor a 80 colonne e se è stato inserito il modo 80) continuerà a presentare lo schermo di testo a 80 colonne. Per provare questo basta inserire i due comandi GRAPHIC 5: GRAPHIC 1. Se poi inseriamo PRINT RGR(0) otterremo il valore 6.
La System Guide è particolarmente lacunosa per quanto riguarda questi aspetti. In particolare, la voce dell'Encyclopaedia dedicata alla funzione RGR omette i due valori:
6 schermo grafico (GRAPHIC 1) con testo a 80 colonne
8 schermo grafico (GRAPHIC 3) con testo a 80 colonne

Le tre uscite video disponibili sul C-128 sono quindi:

- RF (radiofrequenza) per televisore a colori o in bianco e nero: si collega all'antenna del televisore.
- PAL (video composito) per collegamento al monitor o al TV (se questo ha un ingresso monitor o SCART). Un TV collegato in questo modo dà in genere risultati molto migliori.
- RGBI per monitor con funzionamento in modo RGBI.

I primi due servono solo per lo schermo a 40 colonne. La maggior parte dei monitor RGBI è invece in grado di funzionare anche in PAL, cioè a 40 colonne, ma alcuni non lo sono. Un acquisto errato significherebbe rinunciare quindi a tutte le possibilità grafiche del C-128, anche se al limite si potrebbe usare il TV di casa.
Per usare un monitor RGBI diverso dal 1901 Commodore sarà forse necessario acquistare o far costruire un cavetto speciale. Bisogna informarsi presso il rivenditore, anche se, al momento in cui si scrive, è purtroppo possibile che il rivenditore non ne abbia la più pallida idea.

9.3 Maiuscolo, minuscolo e caratteri grafici

Sullo schermo a 80 colonne esiste una grande ed interessante novità, che in parte compensa l'assenza della grafica vera e propria. Con 40 colonne, si hanno due modi di funzionamento per i tasti: "maiuscolo e grafica" oppure "maiuscolo e minuscolo". Si commuta tra i due modi premendo COMMODORE e SHIFT insieme, oppure usando CHR$(14) e CHR$(142), ma la commutazione comporta il cambiamento del testo già sullo schermo. Se quindi abbiamo scritto in maiuscolo e minuscolo, le lettere minuscole diventano maiuscole, e le maiuscole diventano caratteri grafici. Sullo schermo a 80 colonne ciò avviene solo per i caratteri battuti dopo la commutazione. Si possono quindi mischiare i due modi sullo stesso schermo. Ciò rende disponibile un totale di 512 caratteri contemporaneamente.

9.4 Lo schermo inverso

L'Appendice I della System Guide riporta i codici CHR$, CTRL e TAB che hanno effetto sullo schermo a 80 colonne. Quando esistono, i codici CTRL sono in genere più facili da usare dei codici CHR$, almeno per sottolineature con CTRL-B oppure con CHR$(2) e per il lampeggio con CTRL-O oppure CHR$(15).
I codici ESC si inseriscono in modo diretto premendo prima ESC e poi il tasto indicato: in programma si usa la forma

PRINT CHR$(27);"*carattere*"

Quindi per usare il video inverso da programma si usa

PRINT CHR$(27);"R"

Per evitare di scrivere molte volte PRINT CHR$(27), si può stabilire all'inizio del programma:

0 e$=chr$(27)

Il dischetto che accompagna questo libro comprende tre brevi programmi illustrativi, 80-COL, 80.NEG e W2.
Il primo illustra tra l'altro i vari aspetti che si possono dare al cursore:

- normale (rettangolare)
- normale non lampeggiante
- un carattere di sottolineatura (_) lampeggiante
- un carattere di sottolineatura non lampeggiante.

Di questi, il secondo e il terzo sono utili alternative al cursore normale: il cursore piccolo e fermo offerto dall'ultima possibilità è spesso difficile da vedere. Il secondo è disponibile anche sullo schermo a 40 colonne.
Come abbiamo visto, premere ESC e poi R inverte il campo dello schermo.
Gli altri due programmi illustrano l'uso dello schermo negativo e delle finestre. È alquanto complesso descrivere tutti i possibili effetti, ed è perciò consigliabile fare qualche prova in modo diretto.
È probabile che l'esperienza nella programmazione per lo schermo a 80 colonne porterà alla scoperta di tecniche più sofisticate.
Tenendo presente che per ora sono pochi i possessori di un C-128 che hanno anche il monitor adatto, è stato necessario mantenere nei programmi sul disco la compatibilità con il display a 40 colonne. Tutti i programmi non grafici possono funzionare sullo schermo a 80 colonne, ma in genere si è creata su quest'ultimo una finestra contenente 40 colonne. In questo modo l'organizzazione delle righe e la divisione delle parole restano valide su entrambi gli schermi. Una grande eccezione è DIRTUT: questo programma (descritto altrove) elenca il direttorio del disco in modo diverso sui due schermi, ottenendo una lista di file fino a 40 sullo schermo a 40 colonne, e il doppio su quello a 80 colonne. Sul Menu Comandi Vari del Lato 1 del disco, anche i programmi TYPE/MORE e FIND funzionano molto meglio in modo 80 colonne. Poiché sono piuttosto lenti, usano l'espressione IF RGR(0)=>5 THEN FAST, ottenendo così il vantaggio di presentare più informazioni sullo schermo e di funzionare più velocemente.
Chi, a differenza dell'autore di questo libro, non ha la necessità di tenere in considerazione gli utenti che dispongono soltanto dello schermo a 40 colonne, farà molto bene a formattare ogni programma non grafico per lo schermo a 80 colonne.
I programmi del disco avrebbero potuto contenere routine di stampa diverse per i due schermi, ma questo li avrebbe resi più lunghi e lo spazio sul disco sarebbe stato insufficiente.

Quando il C-128 si accende in modo 80 colonne, lo schermo è nero e i caratteri cyan. Premendo ESC-R si ottiene quindi uno sfondo cyan con una cornice nera del tutto simile alla solita cornice dello schermo a 40 colonne, e i caratteri sono neri.
Premere ESC-N per tornare allo schermo iniziale, e provare:

COLOR 6, 2 (colora lo sfondo di bianco)
COLOR 5, 1 (colora il carattere di nero)

Ora inseriamo qualche riga di testo. Poi premiamo ESC-R. Lo schermo diventa nero e il carattere bianco. Inoltre appare una cornice bianca. Inseriamo una dozzina di caratteri dopo aver premuto CTRL-9 (RVS ON). Questo carattere, come già sul C-64, offre l'inversione dei singoli caratteri.
Ora abbiamo anche i caratteri neri, ma sono visibili perché sono circondati di bianco. In altre parole i bit a 0 danno bianco anziché nero.
Battere CTRL-0 per terminare i caratteri inversi ed ESC-N per tornare allo schermo normale. I caratteri che erano in RVS si invertono e sono bianchi su sfondo nero: per il resto lo schermo è bianco con caratteri neri e senza cornice.
Cambiamo il carattere in giallo con COLOR 5, 8 oppure CTRL-8. Inseriamo di nuovo ESC-R. Portiamo il cursore in basso con la freccia e facciamo scrollare lo schermo verso l'alto: le nuove righe sono gialle: se insistiamo tutto lo schermo diventa giallo.
Se inseriamo COLOR 5, 3 il carattere diventa bianco su rosso, se usiamo le frecce le nuove righe che appaiono sono rosse.
Siamo in modo ESC-R. In questo modo, se battiamo COLOR 6, 4 il bordo assume il colore cyan, e il resto dello schermo resta come prima.
Tutto ciò non è affatto semplice. L'unico modo per imparare è di fare dei tentativi in modo diretto e in programma. Vale senz'altro la pena, perché si possono ottenere effetti molto belli.
Anche per fare delle prove sarà più semplice scrivere un programma per poter modificare le righe senza doverle ribattere interamente. Si tenga presente che ESC=CHR$(27). Ecco un modo economico di usarlo

```
10 r$ = chr$(27)+"r": n$ = chr$(27)+"n"
20 ? r$
30 ? "abcdefghijklmnopqrstuvwxyz"
40 sleep 5
50 ? n$
```

Basta cambiare il colore (COLOR 5 per il carattere e COLOR 6 per lo sfondo quando lo schermo non è negativo, e viceversa quando è negativo), dare il RUN, e si vedrà l'effetto. Aggiungendo

```
25 ? "S"
```

dove "S"=CLR, si vede forse meglio il risultato. Dopo 5 secondi il programma restaura lo schermo originale.
Non a caso i programmi dedicati allo schermo ad 80 colonne fanno largo uso di finestre: il comando WINDOW, per quanto utile con 40 colonne, diventa uno strumento di lavoro estremamente prezioso con lo spazio maggiore offerto dallo schermo RGBI. Infatti, mentre sullo schermo più piccolo i problemi riguardano generalmente la difficoltà di inserire tutto il testo che si deve presentare (in particolare nei programmi didattici come quelli sul disco), sullo schermo più grande ci si trova piuttosto davanti alla necessità di una presentazione più ordinata, con la creazione di spazi, colori e blocchi di testo che invitino l'occhio a leggere.

9.5 WINDOW

WINDOW è disponibile sia in modo diretto sia da programma. In modo diretto, se in un dato punto dello schermo si preme ESC-T, e poi si porta il cursore più a destra e/o più in basso e si preme ESC-B, viene creato uno schermo ridotto, o *finestra*, in cui è possibile scrivere, e che si può ripulire, senza disturbare ciò che si trova sul resto dello schermo. Quando si vuole listare un programma, tenendo qualche riga costantemente sullo schermo per un confronto, basta listare o scrivere la riga o le righe in questione, poi creare una finestra in un'altra parte dello schermo e listare il programma lì. Purtroppo questo sistema ha una limitazione: se la riga di BASIC che è stata tenuta sullo schermo al di fuori della finestra supera gli 80 caratteri, non è possibile modificarla, infatti, quando si preme RETURN il sistema operativo non riconosce più i caratteri dopo l'ottantesimo e tronca la riga. Se quindi la riga è lunga è indispensabile listarla di nuovo all'interno della finestra o dopo aver abolito la finestra premendo due volte il tasto HOME.
La versione ESC-T può essere usata in programma, e presenta in un solo caso un grande vantaggio rispetto a WINDOW. Non è necessario stabilire l'angolo inferiore destro della finestra.

```
100 print chr$(27);"t"
```

può essere usato per conservare sullo schermo qualsiasi numero di righe già stampate. ESC-B o CHR$(27)"B" può stabilire una zona in fondo allo schermo. Il resto dello schermo diventa lo schermo effettivo fino a quando non si inserisce un PRINT" seguito da due HOME.

```
200 print"ssS": rem pulisce schermo e abolisce finestra
```

("ss" corrisponde a due HOME, "S" a un CLR).

Con WINDOW, come con CHAR, si possono utilizzare variabili per definirne il formato.
Soltanto gli sprite non sono influenzati da WINDOW: continuano a viaggiare per tutto lo schermo a 40 colonne.

9.6 Creazione di maschere

Esistono con le finestre diversi modi di dividere lo schermo in parti fisse e parti variabili, particolarmente in zone "interattive". Tipiche applicazioni professionali di questo genere possono essere osservate per esempio negli uffici di molti enti (azienda del gas, luce, telefono, anagrafe), dove l'impiegato allo sportello deve consultare rapidamente e frequentemente la situazione di un cliente. Oltre ad avere tasti funzione fissi per le varie operazioni, è normale che la prima (o ultima) riga dello schermo sia una finestra che contiene promemoria del tipo:

F1 TOTALE   F2 ZONA   F3 TARIFFA   F4 CONTRATTI....

Questa finestra resta sempre sullo schermo mentre il resto dello schermo viene occupato da maschere o moduli diversi a seconda della funzione selezionata. La maschera consente di spostare il cursore solo in certe zone, e di inserire nella zona (o campo) in questione soltanto informazioni di determinati tipi. Così ci saranno campi solo numerici, solo alfabetici, e così via. La maschera può essere creata da un comando simile a WINDOW, ed è sostenuta da sezioni di programma che controllano che l'impiegato non inserisca per errore il numero di telefono nel campo riservato al cognome, e così via.
Ma non è necessario scrivere un programma per un'esattoria per trovare utile la possibilità di "disciplinare" l'uso dello schermo.
Il computer spesso deve interrogare l'utente, e si usano vari comandi (GET, GETKEY, INPUT, JOY, POT, PEN) per ottenere le informazioni. È frequente, quando si devono ricevere stringhe o dati numerici, la necessità di evitare che l'utente vada troppo veloce, che sposti il cursore in direzioni errate, e così via.
La soluzione ideale è dividere lo schermo in varie zone, nelle quali appunto l'utente del programma può scrivere, e in altre zone in cui il cursore non può arrivare. Queste possono quindi contenere indicazioni non cancellabili. È possibile creare moduli veri e propri.

SINTASSI DEL COMANDO WINDOW

WINDOW *x1, y1, x2, y2, clr*

Qui *x1, y1* determinano l'angolo superiore sinistro della finestra e *x2, y2,* l'angolo inferiore destro.
Il parametro *clr* invece, se è 1, pulisce lo schermo all'interno della finestra: se è 0 o non è indicato, non lo ripulisce.
Le analogie tra i modi di posizionare WINDOW e CHAR suggeriscono subito l'uso del comando CHAR per le scritte che devono essere stampate in posizioni particolari rispetto alle finestre (nel programma che segue, le indicazioni "Nome", ecc.). Poiché le forme sono uguali (solo che CHAR usa solo *x1, y1*), è facile, in un programma con molte finestre, usare le stesse variabili per posizionare sia la finestra che le scritte.
Il disco non contiene programmi che realizzino fino in fondo queste possibilità. Il programma WINDOW crea uno scheletro per l'input di dati anagrafici in zone fisse dello schermo. Per utilizzarlo in un'applicazione reale sarebbe necessario aggiungere un meccanismo per consentire all'utente di spostarsi da una finestra all'altra, anche dal basso verso l'alto.

```
window.list
10 color 0, 13: color 4, 1: color 5, 1
20 print"S"
30 open 3, 0
40 char , 1, 5, "rNomeR[\r                 RP]"
50 char , 10, 9, "rEta´R[r     RP]"
60 char , 15, 15, "rINDIRIZZORP"
70 char , 1, 18, "rVia[r                        RP]"
80 char , 1, 20, "rNo.R[r      RP]"
90 char , 10, 22, "rCitta´R[r                RP]"
100 print "sq     rPREMERE UN TASTO PER INIZIARER": getkey a$
110 window 6, 5, 21, 6, 1
120 input#3, n$
130 window 15, 9, 19, 10, 1
140 input#3, e$
150 window 5, 18, 37, 19, 1
160 input#3, i$
170 window 5, 20, 10, 21, 1
180 input#3, nc$
190 window 17, 22, 37, 24, 1
200 input#3,c$
210 print "ss"
```

Le righe seguenti sono quelle relative alla voce "Eta'".

```
50  char , 10, 9, "Eta´[     ]"
130 window 15, 9, 19, 10, 1
140 input#3, e$
```

Sono stati rimossi i caratteri speciali che indicano il colore, il negativo, ecc., perché sia più facile capire il rapporto tra le coordinate e il numero dei caratteri. La finestra appare sulla riga 9 (e finisce sulla 10, perché altrimenti la risposta dell'utente non resterebbe visibile dopo aver premuto RETURN, ma si utilizza solo la riga 9). In un'applicazione più pratica sarebbe meglio evitare questa soluzione e usare un CHAR per inserire la risposta dell'utente nella finestra, subito prima di passare alla prossima finestra. L'uso di CHAR in questo programmino è più limitato; nella riga 50, CHAR mette i 5 caratteri Eta' [prima della finestra e quindi, mentre CHAR parte da $x=10$, WINDOW parte da $x=15$. Finisce a 19, mentre il CHAR mette nella colonna 20 la chiusura di campo]. I caratteri speciali r e R denotano RVS ON e RVS OFF: questo permette di ottenere un campo colorato diversamente per ciascuna indicazione da inserire.
La riga 210 (PRINT"ss") corrisponde a due HOME che, come si è già detto, rappresenta un modo di ripristinare lo schermo normale. È una buona norma inserire un'espressione del genere almeno alla fine del programma, per evitare che un nuovo programma venga influenzato da una finestra precedente, anche se questo può servire; sul dischetto di questo libro, vari programmi, se usati in modo 80 colonne, emulano il modo 40 con una WINDOW che non è sempre presente nei singoli programmi, perché a questi programmi si arriva dai menu, che a loro volta si servono di una finestra di 40 colonne. In altre parole, il caricamento di un programma nuovo non annulla la finestra esistente.
Infine, l'uso speciale di INPUT qui deriva dalla semplice esigenza di inputtare una stringa senza avere un cursore lampeggiante. OPEN 3,0 apre la tastiera (Device 0) come file e INPUT#3 funziona normalissimamente ma senza il cursore visibile.
WINDOW è un termine che viene usato anche con riferimento agli schermi GRAPHIC 2 e GRAPHIC 4. Queste due specifiche creano una finestra di cinque righe di testo normale in fondo allo schermo grafico; il numero delle righe può essere variato con l'ultima opzione: si veda la voce GRAPHIC nel Dizionario del BASIC, e la discussione nei due capitoli precedenti.
Il grande vantaggio dello schermo a 80 colonne è il maggior spazio disponibile per le scritte, e quando una certa quantità di spazio rimane libero è possibile usarlo sia per creare cornici ornamentali, sia per presentare indicazioni utili per l'utente. Tra queste possibilità esiste l'uso dello schermo negativo, illustrato nei programmi 80.NEG e W2.
Dopo aver creato una finestra, è possibile ottenere vari effetti con ESC-R. In primo luogo non è vero che non si può ottenere una cornice sullo schermo ad 80 colonne: ESC-R normalmente ne crea una. Le varie combinazioni sono numerosissime e l'unico modo di capire è con la sperimentazione diretta, creando una finestra qualsiasi (per esempio caricare il Menu Generale del dischetto e premere 1 per terminare: si trova una finestra già impostata). Poi provare tutte le molteplici combinazioni di COLOR 5, *n* e COLOR 6, *n* insieme a CTRL-9 e CTRL-0.

10

Espansioni di memoria: FETCH, STASH, SWAP

Chi ha mai scritto un programma per il VIC-20 o per il Commodore 16 avrà una chiara immagine dell'espansione di memoria come estensione della RAM di sistema, cioè della memoria per programmi e variabili. Nel C-16, per esempio, un programma grafico non può superare una lunghezza di 2 K perché tutto il resto della memoria è occupato dallo schermo grafico. Già nel caso del C-64 l'esigenza di ampliare questa memoria è meno sentita. Per molte applicazioni è più comodo raggruppare programmi più brevi in "overlay", cioè in modo che ciascun programma carichi ed esegua un altro. I singoli programmi sono peraltro più facili da scrivere e da collaudare. L'unica obiezione contro questa soluzione insorge quando è necessaria una certa velocità: con il drive 1541 o, molto peggio, con l'unità a cassette, i tempi morti possono essere troppo lunghi. In questi casi è naturale pensare di utilizzare un'espansione che consenta di caricare all'inizio un programma molto lungo.
Per questi motivi, se diciamo che il C-128 può essere espanso fino a 256 K o addirittura fino a 640, qualche lettore si domanderà chi potrà mai scrivere un programma di più di 500 K.
Nel Capitolo 2 abbiamo suggerito che scrivere programmi molto lunghi per il C-128 sarebbe meno efficiente che scrivere una serie di programmi più brevi usando il sistema dell'overlay. Le espansioni non danno comunque la possibilità di scrivere programmi più lunghi, ma servono come banca dati, con la possibilità di accedere ai dati quasi istantaneamente: consentono quindi di utilizzare il sistema dell'overlay con il vantaggio della velocità, anche usando un drive lento come il 1541.
La voce BANK nel Dizionario illustra come la memoria è divisa in ban-

chi: il banco 0 è riservato ai programmi in BASIC, mentre il banco 1 contiene le variabili. Mentre i banchi 4, 5, 8, 9 e 12-15 servono al sistema operativo per diversi scopi, restano inutilizzati i banchi 2, 3, 6, 7, 10 e 11. Questi sono riservati per le espansioni. Il Modulo di Espansione RAM #1700 offre 128 K di RAM addizionale, mentre il #1750 fornisce ben 512 K, portando la RAM totale rispettivamente a 256 K e a 640 K.
Se non servono direttamente per programmi, a che cosa sono destinate le espansioni? Forse il modo più comprensibile di considerarle è come periferiche, come se fossero un'alternativa al disk drive, al registratore o alla stampante.
Servono per immagazzinare grandi quantità di dati che possono essere utilizzati molto rapidamente grazie a FETCH, STASH e SWAP, che servono rispettivamente per caricare dall'espansione, salvare nell'espansione e scambiare dati con l'espansione. Per la sintassi si veda il Dizionario. Il seguente esempio salva in espansione i 9024 byte di informazioni relativi a uno schermo grafico:

```
STASH 9024, 7168, 0, 3
```

Questo dice in pratica "salvare i 9024 byte che cominciano all'indirizzo 7168 nel banco già selezionato (0) a partire dall'indirizzo 0 del banco 3". Il FETCH per richiamare nella memoria di sistema gli stessi dati sarebbe:

```
FETCH 9024, 7168, 0, 3
```

È difficile immaginare una sintassi più semplice. Nonostante ciò, esiste un modo ancora più semplice e più efficiente di utilizzare le espansioni RAM.

10.1 Il RAM Disco

Supponiamo di caricare nell'espansione l'intero contenuto di 3 dischetti (formato 1541) e poi, mediante un apposito programma, di dire al calcolatore di trattare l'espansione esattamente come se fosse un disk drive addizionale. Usata in questo modo l'espansione viene chiamata RAM Disco. Per il calcolatore è esattamente come se fosse un'unità periferica (U9, per esempio) con la differenza di essere 150 volte più veloce del 1541, una dozzina di volte più veloce del 1571 e 5 volte più veloce del 1581.
Il sistema GEOS del C-64 ha una possibilità che rappresenta l'esatto contrario di questo metodo: per sopperire alla mancanza di RAM libera nel calcolatore, è capace di creare dei file su disco che vengono letti esattamente come se fossero parte della memoria centrale. È un sistema lento

che consente però di superare i limiti della RAM del C-64. Con il C-128 espanso la situazione è rovesciata: esiste RAM in abbondanza, ed è il disco che è lento e limitato.
Il dischetto che accompagna questo volume consente di vedere, per esempio, la velocità di caricamento (con BLOAD) di un intero schermo grafico: la velocità precisa dipende quasi esclusivamente dalla velocità del drive. Si tratta in tutti i casi di un file di 9 K. Se all'inizio di una sessione di lavoro carichiamo dal disco una cinquantina di questi bit map (un totale di 450 K di dati) possiamo usare FETCH per visualizzarli rapidamente, oppure, grazie al programma che crea il RAM Disco, usare i normali comandi (BLOAD, DLOAD...) del disco, poiché questa RAM sarà per il calcolatore a tutti gli effetti un disco. BLOAD è già veloce, ma buona parte del tempo che impiega riguarda la lettura del floppy disk. Con il RAM Disco è quasi istantaneo.
Non è una grave distorsione della verità dire che il caricamento sarà istantaneo: la velocità di trasferimento dei dati con FETCH e STASH è infatti di 1 MB al secondo.
Con questo sistema potremmo quindi creare anche grafici animati (un paesaggio con un fiume che scorre, per fare un esempio, potrebbe richiedere 4 o 5 file soltanto). Ma naturalmente esistono applicazioni più serie: in primo luogo il caricamento di programmi alla stessa velocità praticamente istantanea.
Questo apre la strada verso applicazioni che, per questioni di velocità, nessun programmatore si sarebbe mai sognato di tentare con il C-64. Con 640 K di memoria, e con il BASIC 7.0 per scrivere i programmi (che se necessario possono essere convertiti in linguaggio macchina tramite un compilatore) sono possibili applicazioni professionali degne di una piccola azienda o studio professionale. Il BOOT del sistema crea automaticamente il RAM Disco: l'operazione può richiedere qualche minuto, ma l'intervento umano si limita all'accensione e forse alla sostituzione di un disco. Per il resto della giornata il calcolatore sarà già pronto per eseguire una vasta gamma di programmi istantaneamente. Naturalmente, oltre al RAM Disco, resta disponibile il drive vero, e quindi le operazioni non sono limitate ai dati già in memoria.
Le due espansioni sono vendute con un disco illustrativo. Sul primo lato (modo 128) sono comprese alcune dimostrazioni impressionanti di ciò che si può fare con il BASIC: il secondo lato contiene invece un nuovo disco CP/M che sostituisce interamente quello già fornito insieme al C-128 inespanso. La differenza consiste nel fatto che si è creato un drive M, che è in realtà l'espansione di memoria. Si può quindi avere l'intero sistema operativo in memoria: soprattutto chi dispone soltanto del drive 1541, che per il CP/M è di una lentezza esasperante, potrà lavorare 150 volte più velocemente. Per copiare dal disco al RAM Disco in modo CP/M basta il comando:

PIP M:A:*.*

Il RAM Disco per i C-128 in modo C-128 non era ancora disponibile in Italia durante la stesura di questo capitolo.

10.2 CD-ROM: il disco laser

Il Compact Disk (CD) offre altre possibilità nel campo delle ROM (memoria a sola lettura). Si tratta dello stesso tipo di disco già molto diffuso per l'ascolto della musica e che, com'è noto, utilizza una registrazione di tipo digitale (numerico) e non analogico come il fonografo originalmente inventato da Edison. Usando lo stesso sistema è possibile registrare dati leggibili da un calcolatore. La capacità di un solo disco supera le necessità di quasi ogni utente. Raggiunge infatti i 550 megabyte, e questo significa per esempio che un solo disco può contenere 1000 copie di questo libro (250.000 pagine circa), oppure un'enciclopedia in 20 volumi, o ancora l'intero elenco telefonico italiano (calcolando una stringa di 90 byte per ogni abbonato).
Attualmente si parla di CD-ROM, ma è probabile l'introduzione di sistemi di CD-RAM che consentano all'utente di scrivere sul disco, offrendo un sistema di conservazione dei dati molto più durevole degli attuali supporti magnetici.
In ogni caso, per la prima volta nella storia umana, il libro ha un serio rivale: per continuare con l'esempio dell'enciclopedia, un solo disco, oltre al testo, può contenere anche il software globale necessario per leggere le voci, per consultare l'indice analitico e per stampare su carta i testi. E potranno essere presenti, naturalmente, anche le illustrazioni, eventualmente animate. Più importante ancora, sempre grazie a un programma che può risiedere sullo stesso disco, una voce può leggere i testi. Ciò significa un vantaggio incalcolabile non soltanto per i non vedenti.
Al momento in cui si scrive questo capitolo è appena iniziata la diffusione di lettori CD per personal computer, e non è possibile prevedere la data in cui saranno disponibili per calcolatori come il C-128. Non esistono comunque motivi per dubitare che siano utilizzabili con il C-128 o per pensare che questo calcolatore abbia particolari limitazioni rispetto a un personal più costoso.

11

Gli sprite

Un termine usato dalla Commodore per definire questi fenomeni è *programmable movable object blocks*: blocchi-oggetto programmabili e movibili. *Sprite* in inglese deriva dalla parola *spirit*, e si usa (nelle fiabe, per esempio), per definire quegli esseri eterei, come le fate, che popolano le leggende.
In sintesi, uno sprite è una forma grafica che occupa sullo schermo un'area rettangolare grande 24×21 pixel (pari a 3×3 caratteri sullo schermo a 40 colonne).
Può essere presente, in movimento oppure fermo, sullo schermo di testo a 40 colonne, oppure sui vari schermi grafici, ma non sullo schermo a 80 colonne.
Fino a 8 sprite possono essere conservati simultaneamente in un'area di memoria dedicata esclusivamente agli sprite. Ma, come vedremo più avanti, anche se si possono visualizzare 8 sprite per volta, se ne può conservare un numero qualsiasi usando delle stringhe, e questi possono essere visualizzati in tempi abbastanza rapidi da poterli usare per effetti di animazione.
L'utente può creare i propri sprite in vari modi. Tra questi, SPRDEF rappresenta il metodo più sofisticato, mentre SSHAPE consente di copiare dati dallo schermo grafico e inserirli in una stringa e poi in uno sprite. SPRSAV, invece, consente di inserire i dati di uno sprite in una stringa e, viceversa, di inserire i dati di una stringa nella memoria degli sprite.
Molti lettori preferiranno a questo punto caricare il programma tutorial SPRTUT prima di leggere oltre: renderà molto più facile capire le descrizioni delle prossime pagine.

Uno sprite ha (oltre a un colore di sfondo) un colore principale sempre presente e due colori addizionali (multicolor).
Uno sprite può essere reso visibile con il comando SPRITE e fatto muovere in tutte le direzioni (oppure collocato, fermo, in un punto qualsiasi dello schermo) con MOVSPR.
Uno sprite può essere salvato in memoria con SPRDEF o con SPRSAV, mentre per salvarlo su dischetto si utilizza BSAVE.
Il comando SPRDEF consente di:

- disegnare uno sprite servendosi di uno schermo speciale, quasi un menu
- di colorarlo e osservarlo sia notevolmente ingrandito, sia nella grandezza che avrà quando verrà usato
- di salvarlo in memoria, in modo che possa in seguito essere salvato su dischetto con il comando BSAVE.

11.1 SPRDEF

Se si sta lavorando con lo schermo a 80 colonne e si dà il comando SPRDEF la macchina passa automaticamente a 40 colonne: per evitare confusioni è quindi consigliabile passare a 40 colonne prima di iniziare il lavoro con gli sprite.
SPRDEF elimina anche il contenuto dello schermo grafico (come SCNCLR): usare eventualmente il programmino SALVAGRAFICI (dischetto di questo libro: Menu Comandi Grafici) se il grafico attualmente in memoria non può essere ricreato con un programma.
Il comando SPRDEF fa apparire un rettangolo (largo 24 quadrati e alto 21) sul lato sinistro dello schermo. Questo rappresenterà il nostro sprite: ognuno dei quadratini è grande quanto un carattere sullo schermo, ma definirà in realtà un solo bit dello sprite che produrremo, che quindi sarà costituito di 24×21 bit. Sotto questo quadrato si trova la scritta:

SPRITE NUMBER?

Si possono definire fino a 8 sprite per volta. Noi battiamo 1, ma non è necessario definire gli sprite in un ordine preciso: potremmo anche cominciare con l'ottavo. A questo punto troviamo che esiste un nuovo cursore (pixel cursor) nell'angolo superiore sinistro del quadrato. Questo non lampeggia ed è a forma di +.
Per definire lo sprite abbiamo le seguenti possibilità:

- Cambiare il colore principale usando CONTROL e COMMODORE come nel modo diretto normale (CONTROL-4 dà cyan: COMMODORE-4 dà grigio scuro, e così via).

- Colorare un pixel nello stesso colore dello sfondo: questo serve per correggere un errore, visto che i pixel all'inizio sono ovviamente tutti nel colore dello sfondo. Premere il tasto 1.
- Colorare un pixel nello stesso colore del cursore (colore principale). Premere il tasto 2.
- Colorare un pixel in multicolor 1. Premere il tasto 3.
- Colorare un pixel in multicolor 2. Premere il tasto 4.

Pixel è il nome che si dà all'elemento minimo d'immagine sullo schermo: in questo caso uno qualsiasi dei 24×21 punti dello sprite.
Lo sprite ha quindi sempre almeno due colori (quello fondamentale e il colore attuale per lo sfondo dello schermo, che è naturalmente il colore delle parti dello sprite che non sono disegnate): l'opzione multicolor ne aggiunge altri due.
Per usufruire di più di due colori dobbiamo definire questi colori con SPRCOLOR (si veda anche la voce nel Dizionario del BASIC). Per esempio, SPRCOLOR 5, 6 darà il viola (5) sul tasto 3 e il verde (6) sul tasto 4. I numeri sono i soliti numeri per i colori (v. COLOR). SPRCOLOR non può essere usato mentre si sta usando SPRDEF: sarà necessario quindi definire i 2 colori prima (oppure dopo).
Lo sprite, però, avrà solo i colori 1 e 2 se non premiamo il tasto M (per multicolor). E, purtroppo, come con gli schermi grafici, il multicolor si paga con una definizione minore.
Da qui in avanti, suggerisco che il lettore tenga acceso il calcolatore e che provi in proprio le operazioni: sarà infatti molto più facile capire gli effetti che si ottengono.
Se si tiene premuto uno dei tasti 1 2 3 4, il cursore + continua a spostarsi. Se non si desidera che questo accada, basta premere A, mentre per riattivare la funzione, basta premere A di nuovo.
Premendo RETURN il cursore + va all'inizio della riga successiva HOME (senza SHIFT) porta il cursore + all'angolo superiore sinistro.
CLR+SHIFT elimina tutto il lavoro fatto finora, pulendo il quadrato e riportando il cursore + all'angolo superiore sinistro.
Ciò che è importante per ora è soltanto ottenere 4 colori diversi. Se i colori non sono quelli voluti si possono cambiare anche più tardi. Quando si passa a usare effettivamente lo sprite si possono usare i comandi COLOR, SPRITE e SPRCOLOR per definire rispettivamente i colori dello sfondo dello schermo, del tasto 2 e dei tasti 3 e 4. Per ora stiamo solo definendo le varie parti.
Quando è stato premuto il tasto M, il cursore + si muta in ++. Questo è perché in modo multicolor si definiscono due pixel per volta in senso orizzontale, esattamente come succede con GRAPHIC 3 e GRAPHIC 4 (si vedano i capitoli sulla grafica). Premendo M di nuovo si torna come prima.

Nella parte destra dello schermo, man mano che si accendono nei vari colori i pixel dello sprite, compare lo sprite stesso, cosicché si ha un'idea più precisa di ciò che si sta disegnando. Appare nelle dimensioni che avrà sullo schermo e con i colori attualmente definiti.
Ora proviamo a premere X. Si vedrà che lo sprite visualizzato sul lato destro dello schermo si allunga in senso orizzontale: premendo invece Y si allunga in senso verticale. Anche in questo caso *non* stiamo definendo lo sprite in modo irrevocabile: come per i colori, la decisione finale per l'uso viene dopo (usando il comando SPRITE); X/Y servono per capire cosa stiamo ottenendo, e definiscono i default. Quindi, se salviamo lo sprite con espansione X e Y, esso apparirà espanso solo se il comando SPRITE (che, come si è detto, serve per visualizzare lo sprite) non indica il contrario.
Quando abbiamo definito la forma desiderata, questa si può salvare in memoria premendo SHIFT-RETURN. Il calcolatore salva così lo sprite che abbiamo definito. Ma non lo salva sul dischetto: lo fissa nella memoria (RAM) riservata agli sprite: in seguito sarà necessario salvarlo in un file. Lo schermo si pulisce e torna la domanda:

SPRITE NUMBER?

A questo punto se premiamo 1 ci ripresenterà lo sprite 1, e possiamo modificarlo ancora, e poi salvarlo di nuovo con SHIFT-RETURN.
Altrimenti possiamo inserire un altro numero (non necessariamente 2) per definire un nuovo sprite.
Un altro tasto utile è C. Supponiamo di voler usare una parte dello sprite già definito, o di volerlo usare tutto (in certi casi un duplicato può essere necessario: un determinato sprite non può essere presente sullo schermo in due posizioni diverse). Premendo C e poi inserendo il numero dello sprite già definito che vogliamo ricopiare, questo appare sul quadrato. Possiamo poi modificarlo, se necessario, e salvarlo con il nuovo numero.
Se invece non vogliamo definire un nuovo sprite, basta premere RETURN quando arriva la domanda SPRITE NUMBER? per tornare allo schermo normale.
STOP, usato mentre stiamo definendo uno sprite, elimina il lavoro in corso e ci riporta a SPRITE NUMBER?. Lascia quindi intatto lo sprite sul quale stavamo lavorando (se già esisteva), senza tener conto delle ultime modifiche.
SHIFT-CLR, invece, elimina il contenuto dello sprite senza chiederci di dare un nuovo numero.
Riassumiamo quanto si è detto fin qui:

- SPRDEF consente di definire, punto per punto, uno sprite, senza usare formule matematiche o altro, e con la possibilità di vedere costantemente ciò che accade allo sprite.

- Definisce la forma dello sprite e anche se sarà multicolor o meno. Ma le scelte non riguardanti la forma possono essere revocate con altri comandi quando si utilizza lo sprite.
- Lo sprite definito viene salvato in memoria premendo SHIFT-RETURN.
- STOP abbandona lo sprite, che viene perduto se non era già stato salvato, e conservato senza le ultime modifiche se era già esistente.

I tasti da usare nel lavoro sono quindi:

1	Colora il pixel nel colore dello sfondo.
2	Colora il pixel nel colore principale.
3	Colora il pixel in multicolor 1.
4	Colora il pixel in multicolor 2.
CONTROL	Insieme a un numero cambia il colore fondamentale (tasto 2). Se il numero voluto è maggiore di 8 si usa invece COMMODORE.
Frecce	Muovono il cursore (+ o + +): si possono anche usare i tasti CRSR.
HOME	Porta all'angolo superiore sinistro.
CLR	(SHIFT-HOME) ripulisce il quadrato dello sprite.
1 2 3 4	Colorano il pixel (sfondo, principale, multicolor 1, multicolor 2).
STOP	Abbandona il lavoro: non uccide lo sprite esistente.
SHIFT-RETURN	Salva lo sprite in memoria. Porta alla domanda successiva SPRITE NUMBER?
X	Espande lo sprite orizzontalmente.
Y	Espande lo sprite verticalmente.
M	Multicolor.
C	Copia i dati di uno sprite esistente (con un altro numero).
A	Abilita/disabilita la ripetizione dei tasti.

A questo punto abbiamo uno o più sprite salvati in memoria. Ci resteranno fino a quando sarà acceso il calcolatore; ma se lo spegniamo, tutto è perduto. È quindi necessario salvare lo sprite su disco — o meglio tutti gli sprite definiti si salvano su disco — con BSAVE. Si ricuperano più tardi con BLOAD.

11.2 SSHAPE e GSHAPE con gli sprite

Uno sprite può essere definito anche in un modo completamente diverso, disegnando sullo schermo grafico e poi usando SSHAPE e GSHAPE per salvarlo in una stringa o per riutilizzarlo come un carattere qualsiasi,

che si può "stampare" dove apparirà fermo. Questi ultimi due comandi sono trattati nel Dizionario del BASIC. Si può inoltre usare il comando SPRSAV per salvare uno sprite, contenuto in una stringa, nell'area di memoria degli sprite: in seguito potrà essere usato esattamente come gli sprite definiti con SPRDEF, e quindi anche salvato su disco con BSAVE. In genere, qualunque sia il metodo usato per creare lo sprite, converrà usare BSAVE piuttosto che tenere nel programma tutte le istruzioni necessarie per crearlo ogni volta che si esegue il programma stesso.
Il programmino MICROCHAR illustra un uso di SSHAPE con gli sprite: SHAPES (SSHAPE/GSHAPE sul menu) ne illustra altri.
Le stringhe prodotte con SSHAPE o con SPRSAV potrebbero teoricamente esser salvate su disco con un normale PRINT#: è però necessario tener presente che i "caratteri" in queste stringhe non sono normali codici ASCII. Ne risulta che PRINT# e INPUT# daranno risultati imprevedibili e perfino disastrosi. È quindi necessario usare MID$ per separare le stringhe in singoli caratteri, ottenere il numero del carattere con ASC, usare poi PRINT# per scrivere i valori numerici, e il metodo inverso con GET# per ricomporre le stringhe. Il metodo BSAVE è infatti comodissimo, poiché si possono organizzare, accanto a uno stesso programma, anche moltissimi file esterni, ciascuno con i suoi sprite, e poi utilizzarli tutti (anche più volte) per creare numerosi effetti. L'area di memoria degli sprite viene salvata in un file binario di 3 blocchi. Poiché nel direttorio questo file appare con la sigla "prg", i programmi del disco aggiungono il prefisso "sp." ai file di sprite, per non confonderli con i programmi.

11.3 BSAVE

In un altro capitolo è già stato descritto l'uso di BSAVE per salvare l'intero contenuto di uno schermo grafico tramite una routine contenuta in SALVAGRAFICI. Si suggerisce di seguire questo metodo per le stringhe formate con SSHAPE: stamparle sullo schermo grafico con GSHAPE e poi usare la routine di SALVAGRAFICI per ricuperare l'intero schermo. Se per esempio dobbiamo creare un paesaggio che sarà lo sfondo per un videogioco, conviene creare questo una sola volta, lavorando in modo diretto o con un apposito programma, oppure con una combinazione di queste due possibilità. Si salva con BSAVE, e in seguito basta una sola istruzione nel programma per richiamarlo: ma questo non è tutto. Infatti, se supponiamo di dover creare vari schermi simili, con molti elementi comuni, conviene salvare prima uno schermo che abbia solo gli elementi comuni. Si può in seguito richiamare questo schermo anche più volte, aggiungere gli elementi non comuni a tutti gli schermi, salvarlo con un nome nuovo, e ripetere l'operazione per un altro schermo. Questo è in sinte-

si lo stesso metodo usato da Walt Disney per la creazione dei cartoni animati. Non è solo più comodo che ridisegnare lo schermo più volte; ma garantisce anche che tutti gli sfondi siano identici. Uno schermo ricreato usando opzione "1" (load) di SALVAGRAFICI può infatti essere trattato esattamente come il disegno originale, e si può quindi continuare il disegno come se non fosse stato interrotto. Il numero degli schermi di questo genere è limitato solo dallo spazio disponibile sul disco, poiché ciascuno schermo occupa 36 blocchi sul dischetto. Infine, va da sé che gli sprite possono essere usati come elementi mobili anche su schermi salvati in blocco con BSAVE e poi ricaricati con BLOAD.
Si deve tener presente che SALVAGRAFICI non salva gli sprite eventualmente presenti sullo schermo grafico, perché essi vengono da un'area di memoria completamente diversa. Una forma grafica può però essere creata con SPRDEF, salvata in una variabile con SPRSAV e poi inserita sullo schermo grafico con GSHAPE; con questa soluzione diventa parte dello schermo grafico ed è salvabile con SALVAGRAFICI. Si veda il Capitolo 10 sulle espansioni di memoria per le notevoli possibilità che esse offrono in questo tipo di lavoro.
Il programmino SALVASPRITE del Menu Grafico è costruito esattamente come SALVAGRAFICI: in pratica cambiano soltanto il prefisso (sp.) e gli indirizzi di BSAVE. Il programma visualizza gli sprite con i loro numeri, e come SALVAGRAFICI, consente di verificare gli sprite appena salvati.

VERIFY "*nomefile*",8,1

Notare che DVERIFY serve solo per programmi: VERIFY seguito da ,8 (per il disco) e da ,1 (indica di usare l'indirizzo da cui il file è stato salvato) effettua la stessa operazione su qualsiasi area di memoria. Tutti hanno visto gli sprite nei videogiochi, per creare astronavi, omini, e tante altre forme. Non bisogna però dimenticare che è anche possibile creare con questo metodo (e, ciò che conta, rapidamente) caratteri speciali di ogni forma e colore. Questi potranno poi essere usati su schermi di testo a 40 colonne o su schermi grafici, tramite i comandi SPRITE e GSHAPE oppure con il salvataggio in blocco del bit map di un intero schermo, come descritto sopra. In quest'ultimo caso è chiaro come si possono creare lettere dell'alfabeto di qualsiasi forma e grandezza, posizionarli per creare schermi di titoli complessi e salvarli in blocco. Lo sprite, quindi, anche se una delle sue caratteristiche più affascinanti è quella di muoversi, ha anche molto valore come elemento più o meno statico, per esempio per creare titoli.

11.4 MOVSPR

Per capire in pratica come funzionano gli sprite sarà utile il programma tutorial presentato nel prossimo capitolo. Serve però anche una discussione più teorica, e quindi a questo punto passiamo a considerare in particolare i comandi che servono per visualizzare gli sprite (SPRITE), per muoverli (MOVSPR) e per salvarli in stringhe e riconvertire le stringhe in sprite (SPRSAV). Discuteremo anche la funzione RSPPOS, che consente di determinarne la posizione sullo schermo.
In realtà, per vedere il funzionamento dei comandi per visualizzare degli sprite, non è necessario definire uno sprite qualsiasi. Il comando

```
SPRITE 1,1
```

accenderà lo sprite numero 1 in ogni caso, e si vedrà sullo schermo un quadrato grande quanto quattro caratteri.
La specifica SPRITE può essere in realtà molto più complessa, ed è discussa in dettaglio dopo le quattro forme di MOVSPR.
Se il lettore desidera (ed è opportuno) eseguire in modo diretto i comandi descritti in questo capitolo, può creare degli sprite (ne basta uno, in realtà) usando SPRDEF nel modo già descritto. Se preferisce può invece usare gli sprite definiti sul dischetto. Per fare ciò caricare SALVASPRITE (Lato 2 del disco). Poi premere "l" (LOAD): scegliere un file di sprite dal direttorio che si presenterà: il Lato 1 contiene il file "sp.pusher"; basta girare il disco prima di premere "l". Quando il programma termina, premere STOP due volte invece di tornare al menu.
Ora il sistema dispone degli sprite necessari. Da questo momento i comandi descritti nel seguito visualizzeranno come sprite 1 la parte superiore di un robot.
Per muovere uno sprite ci sono quattro comandi possibili, quattro versioni diverse di MOVSPR:

1 specificando angolo e velocità
2 specificando le coordinate x e y (sprite fermo)
3 specificando coordinate relative ($+/-\ x,y$)
4 specificando distanza e angolo di un movimento.

Prima forma di MOVSPR: angolo e #velocità

MOVSPR 1,45 #1

mette in moto lo sprite 1 verso l'alto a destra, a un angolo di 45 gradi e con una velocità di #1 (che è la minima su una scala da #0 a #15). L'angolo specificato determina la direzione del movimento: 0 gradi equivale a 360 gradi. Se il valore è maggiore di 360, non accade un errore: 361=1, e così via.

L'esempio specifica 45 gradi: così lo sprite scompare in alto a destra e ricompare in basso a sinistra, e continuerà a farlo anche se carichiamo un programma (a meno che, naturalmente, il programma non comprenda un comando che dia il contrordine). Il contrordine può prendere due forme:

SPRITE 1,0

Rende invisibile lo sprite ma non abolisce il suo movimento in diagonale attraverso lo schermo.

MOVSPR 1,1 #0

Lo ferma (velocità zero) senza farlo scomparire.

Una volta che abbiamo messo in moto uno sprite con angolo e velocità, l'unico modo di fermarlo è quindi di ripetere il comando con #0. Attenzione: si può specificare qualsiasi angolo con #0, ma non si può omettere l'angolo o inserire uno 0. Se lo sprite è fuori schermo in quell'istante, si fermerà in posizione invisibile, ma continuerà ad esistere. In caso di necessità si può sapere dov'è usando l'espressione:

PRINT RSPPOS(*l*,*n*)

in cui *l* è il numero dello sprite; $n=0$ dà la coordinata orizzontale (x),$n=1$ dà la coordinata y (verticale) e $n=2$ la *#velocità.*

MOVSPR con un angolo e una velocità fa sì che lo sprite si muova all'angolo e alla velocità indicati finché non intervenga un ordine diverso (altro angolo, altra velocità) Quest'altra velocità può essere zero, come abbiamo visto, e ferma lo sprite.

La System Guide non rende facile la comprensione del sistema di coordinate. Gli sprite sono visibili sullo schermo tra

$x=50$, $y=24$ e $x=344$, $y=255$

In queste coppie di valori, la coordinata x è la prima e la y è la seconda, dopo la virgola. Il manuale della Commodore contiene alla pagina 6-22 un errore di stampa che inverte i valori 255 e 344, indicando cioè 255,344 per x e y nell'espressione riportata qui sopra. Questo è in contrasto con lo schemino riportato sulla stessa pagina, che però è a sua volta quasi incomprensibile, di difficile lettura e insufficientemente spiegato nel testo.

Seconda forma di MOVSPR: coordinate *x, y*

SPRITE 1, 1: MOVSPR 1, 100, 100

Questi due comandi visualizzeranno lo sprite 1 fermo alle coordinate 100, 100. Variando uno dei valori (o entrambi) lo sprite salterà alle nuove coordinate.

Se vogliamo collegare uno sprite ad un altro, per esempio a destra dello sprite 1:

SPRITE 2, 2: MOVSPR 2, 124, 100

raggiunge lo scopo, perché uno sprite è largo 24 pixel. Analogamente se volessimo attaccare lo sprite 2 sotto lo sprite 1:

MOVSPR 2,100,121

ottiene questo risultato, perché gli sprite sono alti 21 pixel. Dopo aver fatto questo, se diamo sempre gli stessi comandi MOVSPR per i due sprite, essi si muoveranno insieme: possiamo al limite attaccare insieme tutti gli 8 sprite e muoverli come blocco unico. Ecco come si può creare una astronave lunghissima o un trenino... Oppure un titolo fino a 8 caratteri (più eventuali spazi, basta lasciare più di 24 pixel tra uno sprite e un altro) che si muova in qualsiasi direzione sullo schermo. Se lo sprite è definito con espansione in direzione *x* e/o *y*, i valori 21 e 24 vanno ovviamente aumentati.

Terza forma di MOVSPR: coordinate relative (+/−*x*, +/−*y*)

La terza forma di MOVSPR è simile alla seconda, ma usa coordinate relative, cioè fa precedere ai valori per *x* e *y* un + o un −, e dà luogo a un movimento che parte dai valori precedentemente impostati.
Questa stessa possibilità esiste con DRAW, CIRCLE, BOX e PAINT, anche se il funzionamento è in certi casi piuttosto strano. La System Guide non ne fa cenno.
Non è necessario specificare in questo caso la posizione di *x* o di *y*. Il valore che si specifica viene aggiunto al valore della posizione attuale dello sprite. I vantaggi sono evidenti. Ancora, possiamo usare RSPPOS per avere le posizioni, e determinare i valori da aggiungere (o togliere) in base all'input da tastiera o da qualche altra condizione del programma:

```
5  x = rsppos(0): y = rsppos(1)
10 if x  > 10 or y < 200 then begin
12 :    MOVSPR 1, +10, -10 ..... e cosi´ via.
```

RSPPOS(0) dà la coordinata *x*: RSPPOS(1) la coordinata *y*.

Quarta forma di MOVSPR: distanza;angolo

MOVSPR *n, d; a*

Un'altra variazione relativa: muovere lo sprite *n* a una distanza di *d* pixel nella direzione indicata dall'angolo *a*:

MOVSPR 1,10;45

per esempio, muove lo sprite 1 attraverso una distanza di 10 pixel a un angolo di 45 gradi (cioè verso l'alto a destra).

Chi è interessato a scrivere videogiochi, avrà subito notato come questi quattro comandi, insieme all'istruzione COLLISION e alla funzione

BUMP (la prima è una specie di GOSUB, la seconda dà informazioni sulle collisioni intervenute), possano rendere facile programmare un gioco.

11.5 SPRITE

Questa specifica è quella che visualizza uno sprite, ne definisce il colore fondamentale (l'unico colore quando non è in modo multicolor), stabilisce se ha priorità o meno (cioè se passa davanti o dietro gli altri oggetti sullo schermo (caratteri, grafici), ne stabilisce le dimensioni (espansione nelle due direzioni *x* e *y*), e specifica se si visualizza in modo mono o multicolor.
SPRTUT presenta una stessa specifica SPRITE, variando a uno a uno i parametri per illustrarne meglio gli effetti. Poiché i parametri sono (al massimo) 7, è necessaria un po' di memoria (umana, s'intende) per ricordarli. Non è necessario specificarli tutti, tutte le volte: i default per gli ultimi tre sono quelli specificati durante la definizione con SPRDEF. Ma se vogliamo cambiarne uno solo dovremo mettere tutte le virgole precedenti: così per cambiare l'ultimo (modo), bisogna scrivere: SPRITE*n*,,,,,,*m*

SPRITE *n, a, c, p, ex, ey,m*
n =numero dello sprite
a =acceso (1) o spento (0)
c =colore fondamentale (1-16)
p =priorità (0=sopra i caratteri, 1=sotto)
ex=espansione orizzontale (0=no, 1=sì)
ey=espansione verticale (0=no, 1=sì)
m =modo multicolor=1, normale=0.

Come si è già detto, spegnere uno sprite non ne ferma il movimento e non ne cambia la posizione, così come MOVSPR può piazzare uno sprite fuori dal campo visibile sullo schermo, senza per questo abolirne la visibilità.
Il colore fondamentale è quello definito con il tasto 2, se lo sprite è stato creato con SPRDEF: se invece è stato creato con SSHAPE, corrisponde al colore definito a quel momento con COLOR 1.
La priorità determina se lo sprite passa davanti o dietro gli altri oggetti sullo schermo.
I valori *ex, ey, m* sono stati definiti con SPRDEF: se non specifichiamo niente lo sprite apparirà come definito. Possiamo però cambiare idea con SPRITE (anche più volte).

11.6 Salvare gli sprite

Qui riprendiamo un discorso già iniziato.
Gli sprite occupano una zona di memoria ad essi riservata, che si trova tra gli indirizzi 3584 e 4096 nel banco 0 (si vedano le voci BANK e BSAVE/BLOAD nel Dizionario del BASIC). Questi 512 byte possono ovviamente contenere solo gli 8 sprite attualmente attivi.

```
BSAVE "sp.sprite",B0, P3584 to P4096
```

salva 8 sprite con il nome "sp.sprite". Qualsiasi altro nome va bene, ma è buona cosa usare un prefisso come "sp." perché nel direttorio il file appare come programma, cioè con le lettere "prg". Così possiamo salvare qualsiasi numero di file, ciascuno con 8 sprite (o meno di 8: quelli non definiti restano vuoti, ma questo non crea problemi).
Ma se vogliamo disporre di più sprite già presenti nella RAM del calcolatore, esiste la possibilità di inserire uno sprite in una stringa di 63 byte. Questa stringa può essere rimessa in uno sprite, oppure (con GSHAPE, si veda il Dizionario) la forma dello sprite può essere visualizzata sullo schermo grafico come un disegnino qualsiasi. Il programmino MICROCHAR sul dischetto ne fornisce un esempio.

SPRSAV

SPRSAV prende tre forme per creare e gestire stringhe speciali contenenti informazioni binarie:

- SPRSAV *n, stringa*
- SPRSAV *stringa, n*
- SPRSAV *n1, n2*

Il primo legge i byte nella memoria dello sprite numero *n*, e li copia in *stringa*, senza con questo eliminarli dalla memoria degli sprite. Il secondo fa il contrario: sostituisce nella memoria degli sprite i byte provenienti della stringa al posto di quelli già esistenti per lo sprite *n*.
Il terzo copia (nella memoria degli sprite) i dati per lo sprite *n1*, in modo che lo sprite *n2* ne diventa un duplicato. Il lettore si ricorderà che si può fare lo stesso lavoro usando il comando SPRDEF. Ecco una routine che:

- salva gli sprite esistenti nella matrice A$(*n*)
- carica 8 sprite già definiti dalla matrice S$(*n*), cioè li inserisce nella memoria degli sprite al posto di quelli precedenti.

Nota: per provare questa routine il lettore dovrebbe prima creare 8 sprite e salvarli nella matrice S$ con

FOR T=1 TO 8: SPRSAV T, S$(T): NEXT

e poi crearne altri 8 (oppure caricare altri sprite dal dischetto). In seguito eseguire la routine:

```
10 for t = 1 to 8
15 :  sprite t, 1: movspr t, 45 #10
20 :  sprsav t, a$(t)
30 next
40 :
50 for t = 1 to 8
60 :  sprsav s$(t), t
70 next
```

Gli sprite, accesi e messi in moto nella riga 15, cambieranno forma senza interrompere le loro varie attività.
È quindi possibile, in qualsiasi punto di un programma, caricare con BLOAD una serie di sprite, conservarli in una matrice, caricarne altri 8, aggiungerli alla matrice, e così via. Entro i limiti, s'intende, della memoria del C-128 (fortunatamente ampia).

```
10 bload "sprite 1", p3584 to p4096
15 dima$(255)
20 for t = 0 to 255
30 :  sprsav t, a$(t)
40 :  bload "sprite 2", p3584 to p4096
50 :  sprsav t, a$(t)
60 :  bload "sprite3", p3584 to p4096
70 :....e cosi´ via fino a
99 next t
```

È chiaro che in seguito una routine del tipo:

```
20 for t = 0 to 255
30 :  sprsav a$(t), 1
40 next t
```

cambierà lo sprite 1 256 volte, immettendovi uno dopo l'altro tutti gli sprite salvati.
Questo significa avere a disposizione un cartone animato costituito da 256 fotogrammi. E poiché una matrice può avere più dimensioni, possiamo organizzare delle routine analoghe che diano:

- più di 256 fotogrammi per uno stesso sprite, oppure
- 256 o più fotogrammi per ciascuno di più sprite, ognuno dei quali può quindi muoversi simultaneamente con gli altri.

La velocità di queste operazioni è del tutto soddisfacente, e la metodologia è abbastanza semplice: chi legge queste routine per la prima volta può trovarle difficili da immaginare, ma nella pratica non ci sono problemi.
Il capitolo seguente presenta un tutorial in due forme. La lettura del testo dovrebbe dare le risposte a numerose domande, ma poiché il programma che descrive è necessariamente complesso, il programma stesso è anche un tutorial: descrive se stesso. È quindi consigliabile leggere il capitolo mentre si segue il programma, che è naturalmente disponibile sul dischetto (Menu Grafico).

12

Gli sprite: tutorial

Il programma SPRTUT illustra gli effetti più importanti che si possono ottenere con gli sprite, e dovrebbe offrire al lettore le basi dalle quali partire per produrre programmi anche avventurosi.
Poiché il word processor usato non riproduce i caratteri di controllo (CLR, HOME, colori, frecce, eccetera), se si ricopiasse il programma da queste pagine, il colore e le posizioni delle diciture risulterebbero diversi da quelli ottenuti con la versione sul dischetto.
Lo schermo riproduce i vari comandi: in certi casi tramite un LIST della riga corrispondente nel programma, in altri tramite una simulazione che usa CHAR. Questo consente, molto più facilmente che con PRINT, di inserire nella stringa che viene riprodotta sullo schermo i valori numerici man mano che cambiano (per esempio nelle righe 530-680).
Il programma comprende numerose righe il cui scopo è di rallentarne l'esecuzione per motivi di comprensibilità. I vari comandi si attuano in realtà molto più rapidamente... altrimenti servirebbero solo a chi volesse scrivere programmi per illustrare il loro funzionamento.

12.1 Accendere e spegnere uno sprite

```
35 bload "sp.pusher2"
40 print tab(120) "Il primo passo e´ di creare gli sprite  con
   sprdef (o caricarli  con bload)"
43 print "SPRDEF non e´ presentato sul disco: si  veda il
   capitolo del libro"
```

```
44 print "Abbiamo gia´ caricato 8 sprite in memoria con:"
46 list35
52 print "q": getkey a$
80 print "La riga 110 accende lo sprite 1"
90 list 110
92 print "qq: getkey a$
110 sprite 1, 1
130 print "E si spegne con"
132 list 150
150 sprite 1, 0
160 print "qq: getkey a$
170 print "Accendiamoli tutti:"
174 list190-195
190 for t = 1 to 8
192 :   sprite t, 1, t, 0, 0, 0, 0
195 next
200 print "qq: getkey a$
```

La 110 accende lo sprite 1, fermo verso il centro dello schermo. È nella forma piccola (senza espansioni *x* o *y*), è nero, non è multicolor. Resta sullo schermo per alcuni istanti, poi viene spento dalla riga 150.
Alla fine di questa prima sezione, le righe 190-195 accendono tutti gli otto sprite, usando le prime tre delle 7 opzioni disponibili: la variabile *t* li definisce (va da 1 a 8): l'1 costante li accende e la stessa variabile *t* serve per dargli il colore fondamentale (monocolor), che quindi va da 1 a 8 a seconda del numero dello sprite in questione. Gli zeri successivi sono una precauzione: se qualche altro programma ha già usato degli sprite, essi potrebbero essere espansi, avere priorità 1 o essere multicolor: se invece il calcolatore è appena stato acceso, sono superflui.
A questo punto avremo da qualche parte (sullo schermo o fuori campo) un certo numero di sprite. Il programma continua spiegando che cosa è successo finora.

12.2 Muovere gli sprite con angolo e velocità

```
200 print "qq: getkey a$
210 print "Ci sono 8 sprite ma non li vediamo tutti."
212 print "Bisogna muoverli con"
216 list 230-234
220 sleep 2
230 for t = 1 to 8
232 :   movspr t, 45 #1
234 next
```

Per essere sicuri di vedere tutti gli sprite, un modo è di dare loro un angolo e una velocità: in questo modo (se l'angolo non è di 0, 90, 180, 270 o

360 gradi) tutti passeranno attraverso il campo visibile. Le eccezioni sono i valori che li farebbero passare in direzione verticale od orizzontale: è ovvio che con questi potrebbero restare visibili. L'angolo di 45 gradi è un valore a caso, comunque: 67 o 113, o qualsiasi altro numero — perfino un valore superiore a 360 — servirebbe allo scopo. Più avanti vedremo come posizionare sprite fermi.
Ancora due problemi da risolvere: è difficile leggere le spiegazioni che appaiono sullo schermo, perché gli sprite vi passano sopra. Inoltre, tutti gli sprite si muovono, ma alcuni possono essere sovrapposti agli altri. Poiché l'angolo di movimento è uguale per tutti, se sono sovrapposti resteranno sovrapposti. Con la quarta opzione (0 o 1) del comando SPRITE, possiamo mettere fine al fastidio della lettura, togliendo agli sprite la loro priorità, mentre per evitare che restino sovrapposti l'uno all'altro, o raccolti a grappoli, possiamo imprimere a ciascuno un angolo di movimento diverso.

12.3 Variare i parametri degli sprite

```
240 print "gg: getkey a$
250 print "Ci oscurano le scritte: cambiamo le cose con:"
252 list 270-274
260 print "Ora passano dietro le scritte"
270 for t = 1 to 8
272 :  sprite t, 1, , 1
274 next
280 print "gg: getkey a$
290 print "SPEGNIAMOLI un momento con "
292 list 310
294 list 1670-1690
310 gosub 1670
320 print "gg: getkey a$
330 print "Si muovevano tutti nella stessa direzione.
           Queste righe li muovono in direzioni diverse"
334 list 340-355
340 for t = 1 to 8
342 :  sprite t, 1, t
352 :  movspr t, 40 * t #1
354 next
360 print "gg: getkey a$
380 print "Un po´ di velocita´ con":
382 list 390-394
390 for t = 1 to 8
392 : movspr t, 40 * t #15
394 next
395 print "Il numero 40 * t stabilisce"
397 print "un angolo diverso per ciascuno sprite"
400 print "Il #numero (max. 15) varia la velocita´."
```

Nella riga 272 notare come, non volendo cambiarne i colori, non ci sono parametri tra la virgola dopo ON/OFF (0/1) e quella del colore. Non è obbligatorio inserire tutte le opzioni ma se si omette un'opzione e si vuole specificarne una più a destra, allora bisogna inserire una virgola per ogni opzione omessa. Con 0 (che è il default) per la "priorità", gli sprite coprono le scritte, con 1 passano dietro. Le ultime righe del programma (dalla 1670 in poi) servono per fermare tutto.
La riga 352 usa 40*t per dare a ciascuno sprite un angolo (non importa quale) che va da 40 a 320 gradi: così vengono e vanno in tutte le direzioni.
Infine, la riga 392 è uguale alla 352, soltanto che la velocità viene portata al massimo. Sarebbe ovviamente facile dare a ciascuno una velocità diversa, ma farebbe venir mal di testa a chi guarda lo schermo.

12.4 Posizionare sprite fermi sullo schermo

```
410 print "g": getkey a$
415 gosub 1670
420 print "Ora vediamo i vari modi di muovere gli  sprite,
          sempre usando il comando MOVSPR"
430 list 440
440 sprite 1, 1: movspr 1, 117, 170
450 print "gg: getkey a$ : list 460
460 sprite 2, 1: movspr 2, 141, 170
470 print "g": getkey a$: list 480
480 sprite 3, 1: movspr 3, 165, 170
490 print "MOVSPR con 3 numeri separati da virgole
          accende sprite fermi"
500 print "g": getkey a$: gosub 1670
510 print "movspr n, x, y"
520 print "g": getkey a$
```

Nelle righe 440-480 si visualizzano tre sprite sulla stessa riga (determinata dalla coordinata *y* (170)): le coordinate *x* (117, 141, 165) sono a distanza di 24 pixel l'una dall'altra, in modo che i tre sprite sono incollati l'uno all'altro. A questo punto potremmo usarli come un "supersprite" unico. Il programma illustra questa possibilità in un altro punto: chi volesse vederlo adesso potrebbe fermare il programma premendo STOP e inserire in modo diretto la riga:

```
FOR T=1 TO 3 : MOVSPR T, 90#10 : NEXT
```

```
530 print "Ora osservare questo:"
532 sprite 1, 1
```

```
540 for y = 24 to 250 step 20
542 :   for x = 50 to 344 step 5
550 :      movspr 1, x, y
560 :      a$ = "movspr 1," + str$(x) + "," + str$(y) + "    "
561 :      char , 10, 10, a$
570 :      char , 20, 11, "[x] [y]"
580 next x, y
590 print: print "x determina il movimento orizzontale"
592 print "  y determina il movimento verticale"
600 print "  uno sprite e´ visibile da x = 50, y = 24
            a x = 344, y = 255"
610 print: print "   x va da 0 a 511  y va da 0 a 255"
620 for x = 50 to 344 step 20
622 :   for y = 24 to 250 step 20
630 :      movspr 1, x, y
640 :      a$ = "movspr 1," + str$(x) + "," + str$(y) + "    "
645 :      char , 10, 10, a$
650 :      char , 20, 11, "[x] [y]"
660 :      for t = 1 to 60: next t
662 :   next y: for t = 1 to 13: next t
664 :   for t = 1 to 13: next t
670 next x
680 gosub 1670: movspr 1, 100, 100
```

Questo gruppo di righe fa una cosa più complessa. Appare sullo schermo la riga:

```
movspr 1, 0,  0
         [x] [y]
```

che simula la riga 550 (e più tardi la 630), inserendo però i valori di *x*, *y* cambiati in stringhe tramite STR$. Mentre lo sprite 1 ubbidisce prima a cambiamenti di *x* e poi a quelli di *y* (a STEP di 20 pixel i numeri presentati sullo schermo indicano i valori correnti). Lo sprite fa quindi un certo numero di viaggi in senso orizzontale da sinistra a destra: poi comincia a cambiare *y* invece di *x*, e fa i suoi viaggi in senso verticale.
Si è accennato altrove all'utilità del comando CHAR per questo tipo di visualizzazione: poiché CHAR lavora su coordinate stabilite, non può interferire con il resto dello schermo. Sarebbe molto più complesso usare PRINT e la scritta lampeggerebbe in modo sgradevole. Così, invece, i numeri cambiano come su un orologio digitale.

12.5 MOVSPR

```
690 print "MOVSPR con angolo e #velocita": list 700
700 sprite 1, 1, 3, 1, 1, 1, 1
702 movspr 1, 50, 60
710 for t = 0 to 360 step 10
712 :   a$ = str$(t)
720 :   char , 0, 10, "730 movsprl," + a$ + " #7"
730 :   movspr 1, t #7
740 :   char , 0, 15, "Muove lo sprite a"+a$+" gradi"
742 :   sleep2
750 :   print
752 :   print "Zero e 360 gradi danno lo stesso risultato"
760 :   if t = 170 then for u = 1 to 60: next u
770 :   if t = 0 or t/90 = int(t/90) then sleep 3
780 next t
790 print "Premere un tasto": getkey a$
```

La riga 700 (dopo che il GOSUB 1670 della 680 ha spento lo sprite) rivisualizza lo sprite con il colore rosso (3).
La riga 730 muove lo sprite con angoli diversi di t gradi mentre t varia a step di 10. Ancora una volta la 740 simula la 730 con un CHAR. La 770 dice in pratica "se l'angolo=0, 90, 180, 270 o 360, allora fare una pausa di 3 secondi"; sono i valori che fanno viaggiare lo sprite in orizzontale e in verticale e mi è sembrato opportuno farli restare sullo schermo più a lungo degli altri.
Anche questo pezzo di programma, quindi, dà all'utente la possibilità di vedere direttamente gli effetti di un comando.
Il prossimo spezzone illustra l'uso delle coordinate relative (+/− *x*, *y*): questa volta l'utente è invitato a variare egli stesso la posizione dello sprite: il programma usa prima i tasti + e −, poi X e Y e infine le 4 frecce (in quest'ultimo caso con angolo e distanza relativi).

12.6 Le coordinate relative

```
800 list 810
802 print "lo ferma"
810 movspr 1, 0 #0
815 sleep 10
820 sprite 1, 1, 8, 1, 1, 1, 1
822 movspr 1, 100, 160
830 print "MOVSPR con coordinate relative (+ e -)"
840 print "Premere + o - . Per uscire *"
850 getkey a$
860 if a$ = "+"or a$ = "-" or a$ = "*" then begin: else 850
```

```
870 :      if a$ = "-" then movspr 1, -10, -10:
            char , 0, 10, "movsprl,-10,-10"
880 :      if a$ = "+" then movspr 1, +10, +10:
            char , 0, 10, "movsprl,+10,+10"
890 :      if a$ = "*" then bend: else 850
```

Le righe 860-890 variano di 10 pixel in più o in meno entrambe le coordinate *x* e *y*, a seconda che si prema + o −.

```
900 print "MOVSPR con coordinate relative (+ e -)"
910 print "Premere x o y. Per uscire *"
915 getkey a$
920 if a$ = "x" or a$ = "y" or a$ = "*" then begin: else 915
930 :      if a$ ="x" then movspr 1, +10, +0:
           char , 0, 10, "movspr 1, +10, +0"
940 :      if a$ = "y" then movspr 1, +0, +10:
           char , 0, 10, "movspr 1, +0, +10"
950 :      if a$ = "*" then bend: else 915
```

Nel secondo loop, se si preme X, *x* aumenta di 10: se si preme Y, *y* aumenta di 10. Naturalmente, se si preme il tasto a lungo, lo sprite si muove continuamente.

```
960 print "MOVSPR con distanza e angolo relativi"
970 print "movspr, distanza; angolo"
980 print "Premere una freccia. Per uscire *"
985 getkey a$
990 if a$ = "SU" or a$ = "GIU" or a$ = "DESTRA" or
    a$ = "SINISTRA" or a$ = "*" then begin: else 985
1000 :      if a$ = "SU" then movspr 1, 10;360:
            char , 0, 10, "movspr 1, 10; 360"
1010 :      if a$ = "GIU" then movspr 1, 10;180:
            char , 0, 10, "movspr 1, 10; 180"
1020 :      if a$ = "SINISTRA" then movspr 1, 10;270:
            char , 0, 10, "movspr 1, 10; 270"
1030 :      if a$ = "DESTRA" then movspr 1, 10; 90:
            char , 0, 10, "movspr 1, 10; 90 "
1040 :      if a$ = "*" then a$ = "": bend: else 985
```

Nell'ultimo loop le parole SU, GIU, DESTRA e SINISTRA sostituiscono i caratteri speciali delle quattro frecce.
Questo loop mette infatti a disposizione le frecce. Qui il comando non è più nella forma +/− *x*, *y*, ma come spiega la riga 970, prende la forma:

MOVSPR*n*, *distanza; angolo*

Ogni pressione di un tasto freccia muove lo sprite di dieci pixel a un angolo di 90, 180, 270 o di 360 gradi. Il modo di muovere uno sprite tramite

il joystick è analogo: per una routine che legge il joystick si veda il programma JOYDRAW.

12.7 Tutte le opzioni

Nel lungo loop dal DO nella riga 1072 fino al LOOP della 1330 si variano a una a una tutte le 7 opzioni di SPRITE, tranne il numero dello sprite (che è sempre 1) e la priorità. I CHAR servono a far vedere sullo schermo il valore del parametro che si sta variando in un determinato istante.

```
1050 print "Torniamo a SPRITE"
1052 print "SPRITE numero, colore primario, priorita ,
     espansione x, espansione y, modo
1060 movspr 1, 160, 160: sprcolor  4, 8
1070 print "Per uscire premere *"
1072 do while a$ <> "*": get a$
1080 : sprcolor  1, 1
1090 : char , 10, 11, " c             "
1100 : char , 8, 8, "color E          "
1110 : for t = 1 to 16: get a$
1111 :   if a$ = "*" then 1140
1120 :   sprite 1, 1, t, 1, 1, 1, 0
1122 :   char , 0, 10, "spritel,1,"+str$(t)+",1,1,1,0"
1124 :   sleepl
1130 : next t
1140 : char , 8, 8, "ESPANSIONE X"
1150 : for t = 1 to 10: get a$
1160 :   char , 10, 11, "    x y       "
1170 :   sprite 1, 1, 1, 1, 1, 0, 0
1172 :   char , 0, 10, "spritel,1,1,1,1,0,0    "
1174 :   sleepl
1180 :   sprite 1, 1, 1, 1, 0, 0, 0
1182 :   char , 0, 10, "spritel,1,1,1,0,0,0 "
1184 :   sleep 1
1190 : if a$ = "*" then 1200: else next t
1200 : char , 8, 8, "ESPANSIONE Y"
1210 : for t = 1 to 10: get a$
1220 :   if a$ = "*" then 1260
1230 :   sprite 1, 1, 1, 1, 0, 1, 0
1232 :   char , 0, 10, "spritel,1,1,1,0,1,0 "
1234 :   sleep 1
1240 :   sprite 1, 1, 1, 1, 0, 0, 0
1242 :   char , 0, 10, "spritel,1,1,1,0,0,0 "
1244 :   sleep 1
1250 : next t
1260 : char , 8, 8, "MODO E SPRcolor   "
1262 : a = 0
```

```
1264 : char , 10, 11, "          m"
1270 : for t = 1 to 15: get a$: if a$ = "*"then exit
1280 :   sprite 1, 1, 1, 1, 1, 1, a
1282 :   char , 0, 10, "sprite1,1,1,1,1,1," + str$(a)
1284 :   sleep 1
1286 :   if a = 1 then a = 0: else a = 1
1290 :   sprcolor  t, t + 1
1292 :   if a = 1 then char , 0, 12, "sprcolor " + str$(t)
         + "," + str$(t+1)
1300 : next t
1310 : char , 0, 12, "
1320 : gosub 1670
1330 loop
1335 :
1340 gosub 1670
```

Dopo aver visto lo sprite 1 assumere tutti i colori possibili, lo vediamo espandersi e contrarsi nelle direzioni *x* e *y*. Nell'ultimo dei loop interni (1270-1300), si illustra SPRCOLOR oltre a SPRITE. Il loop accende e spegne la funzione multicolor a seconda del valore di a (0 o 1): per illustrare meglio che cosa sta accadendo, SPRCOLOR nella riga 1290 varia i valori di multicolor 1 e 2 con i valori t e t+1. Poiché le combinazioni di multicolor 1 e 2 ammontano a 16*16 (=256), sarebbe difficile presentarle tutte: qui ne vediamo solo 8.

12.8 SPRSAV e animazione

Dopo uno schermo riassuntivo creiamo i titoli di coda. Questi fanno uso di SPRSAV per creare un cartone animato molto elementare. Tra l'altro vedremo a che cosa serve il mezzo robot che è stato il nostro sprite 1 fino a questo punto. Passiamo anche allo schermo grafico (GRAPHIC 1), se non altro per dimostrare che la cosa non fa nessuna differenza.

```
1600 dima$(11)
1601 color 0, 4
1602 graphic1, 1
1603 color 0, 16: color 1, 3:
     char , 0, 23, "schermo grafico"
1604 bload "sp.pusher2":
     for t = 9 to 11: sprsav t-8, a$(t): next t
1605 print "": bload "sp.pusher"
1606 for t = 1 to 3:
         sprsav t, a$(t): sprsav t+3, a$(t+3):
     next:
```

```
     color 1, 2
1607 movspr 1, 10, 100: movspr 2, 12, 142:
     char , 0, 23, "questa routine usa sprsav e una matrice":
     char , 0, 24, "per creare un´animazione"
```

Per vedere quali sono i nuovi sprite, si può caricare il programma SALVASPRITE e guardare i due file "sp.pusher" e "sp. pusher2". In sintesi, il tronco è, per entrambi i file, negli sprite 1-3, le gambe in 4-6 e la parola FINE in 7-8. Qui stiamo usando SPRSAV con la sintassi:

SPRSAV *nsprite, stringa*$

che salva lo sprite *n* nella stringa A$(*n*).
Una volta che la matrice è preparata, possiamo partire. Ma prima notiamo che SPRSAV costituisce il metodo preferito per le animazioni: visualizzare uno sprite dopo l'altro con SPRITE è possibile, ma il comando è più lento e gli sprite lampeggiano sgradevolmente. Inoltre, con SPRSAV, potremmo avere 8 matrici e animare 8 sprite diversi contemporaneamente.

```
1608 sprite 1, 1, 7, , 1, 1, 0: sprite 2, 1, 7, , 1, 1
1609 sprite 7, 1, 9, 1, 1, 1:  sprite 8, 1, 9, 1, 1, 1:
     movspr 7, 59, 100:  movspr 8, 106, 100
1610 trap 1628: do
1611 :  for t = 1 to 3
1612 :     sprsav a$(t), 1: sprsav a$(t + 3), 2
1613 :     movspr 1, + 1, + 0: movspr 2, + 1, + 0:
           movspr 7, + 1, + 0: movspr 8, + 1, + 0
1614 :     if t = 1 or t = 3 then for f = 1 to 10: next f
1615 :     if rsppos(7,0) = > 160 then exit
1616 :  next t
1617 loop
1618 sprsav a$(9), 1: sleep 1
```

La riga 1608 crea un robot: lo sprite 1 ne è la parte superiore, il 2 la parte inferiore. Il robot spinge davanti a sé il cartello con la parola FINE, contenuto negli sprite 7 e 8. Nella 1612 il rapido cambiamento dello sprite 2 dà il movimento delle gambe, mentre la 1613 sposta lo sprite con coordinate relative da sinistra a destra. Nella 1614 un rallentamento prolunga la visualizzazione delle gambe per t=1 e t=3 (t=2 corrisponde alla posizione intermedia con le gambe verticali; la 1614 evita che questa posizione si veda più delle altre due).
La riga 1615 termina il loop quando gli sprite sono al centro dello schermo. La specifica EXIT manda il programma alla riga 1618, in cui lo sprite 1 viene cambiato nell'ormai familiare robot con il braccio destro alzato.

```
1620 do until rsppos(2,0) = > 340
1621 :     for t = 1 to 3
1622 :       sprsav a$(t + 3), 2: if rsppos(2,0) = > 185 then
             sprsav a$(10), 1: sprite 1, 1, 3
1623 :       movspr 2, + 1, + 0: movspr 7, + 1, + 0:
             movspr 8, + 1, + 0
1624 :     next t: sprite 1, 1, 7
1625 loop: sprite 1, 1, 6
1626 sound 2, 59000, 290, 1, 10, 200, 2
1627 movspr 1, 180 #4: sleepl
1628 for t = 1 to 300: next t:
     movspr 1, 1 #0: sprsav a$(2), 2
1629 sprsav a$(2), 2
```

Si entra ora in un secondo loop, in cui un "errore" di programmazione porta alla tragedia. Mentre il robot si gira per salutare, le sue gambe partono insieme al cartello con la parola FINE. Il robot se ne accorge solo quando la funzione RSPPOS=185. Scompare il suo sorriso a favore della smorfia contenuta in A$(10): diventa rosso lampeggiante (sprite1,1,3 nella riga 1622).
Quando RSPPOS(2,0) informa la riga 1620 che lo sprite 2 è fuori schermo, il tronco del robot diventa verde (nella 1625) e con il classico verso del Gatto Silvestro che cade (1626), la 1627 lo fa precipitare fuori dallo schermo.
Segue la routine (1670-1690) usata in tutto il programma per spegnere e fermare gli sprite: poi le classiche righe di termine programma usate in più o meno tutti i programmi del dischetto.

```
1660 sleep 2: print "g": gosub 1670: print "g":
     graphic 0: goto 2000
1665 rem: spegne e ferma gli sprite
1670 for t = 1 to 8: sprite  t, 0: next
1680 for  t = 1 to 8: movspr  t, 45 #0: next
1690 return
1700 color4, 2: sleepl: color4, 1: return
2000 gosub 1670: scnclr: print "Ripetere/Menu": getkey a$
2010 if a$ = "r" then run
2020 if a$ = "m" then run "gra.menu"
```

Si noti l'utilità della funzione RSPPOS (descritta nel Dizionario) per mantenere velocità e semplicità nei loop.
Nel programmare i cartoni animati, non sono molte le difficoltà se non per quanto riguarda l'uso della matrice, che, come si è detto in precedenza, può essere ben più complessa dell'esempio riportato qui. L'unica difficoltà può essere nella memoria del programmatore, che deve mantenere

un'idea chiara di dove sta mettendo ciascun fotogramma. Sarebbe impossibile in questa sede trattare un esempio più complesso: è chiaro che la matrice potrebbe avere più elementi e più dimensioni, e che il numero degli sprite visualizzati (sopra ne abbiamo solo 4) potrebbe essere maggiore. Ma non cambierebbe niente, se non, appunto, la complessità. È sufficiente mantenere la calma e la lucidità.

13

Suono

Definire le note e gli accordi musicali, anche usando il tradizionale pentagramma, non è un'operazione semplicissima. Oltre all'altezza della nota (che è in fondo la cosa più ovvia) bisogna definirne la durata, e combinarla con le altre note dell'accordo. Poi c'è il volume, il tempo... e così via. Questi sono i parametri che anche Pergolesi e Bach dovevano controllare, e non cambia molto quando si devono scrivere con il calcolatore.
Non sorprende dunque che sui calcolatori più piccoli (ma anche su altri più grossi del C-128) definire anche un semplice motivetto senza accordi e per un solo strumento non sia semplice. Nemmeno il C-128, peraltro, può rendere la musica più semplice di quanto non lo sia in realtà ... ma evita di renderla più complessa del necessario.
Le due voci del BASIC fondamentali per il suono sono PLAY e SOUND. La prima è discussa in questo capitolo, mentre per la seconda si rimanda al Dizionario del BASIC e al tutorial SOUND del disco.

13.1 Le note

In sintesi, PLAY legge una stringa e la "stampa" sull'altoparlante; un po' come la specifica PRINT legge una stringa e la stampa sullo schermo.
Per comporre la stringa, è necessario conoscere un po' la musica, oppure essere dotati di una notevole inventiva. La spiegazione fornita nella System Guide mi sembra valida e chiara. Qui intendo affrontare l'argomento in modo diverso, sperando di rendere il tutto ancora più chiaro.
I nomi delle note musicali usati nei Paesi anglosassoni (e non solo) non

presentano né vantaggi né svantaggi rispetto al sistema italiano. Sono diversi, infatti, solo i nomi delle note:

Do =C	Re = D	Mi = E	Fa = F
Sol = G	La = A	Si = B	

Non ci sono altre differenze importanti.
Il pentagramma è internazionale: si scrivono le note sempre allo stesso modo.
Questa è la scala di Do maggiore, scritta con note intere. È la base per la musica sul computer: ogni nota diversa da queste deve essere segnalata con # (diesis), oppure con $ (bemolle: purtroppo il calcolatore non ha il vero carattere bemolle). La durata delle note deve essere indicata con uno dei seguenti simboli: W (intera); H (mezza); Q (quarta); I (ottava); S (sedicesima); . (puntato - prolunga la nota della metà).

PLAY "W.C HD QE I#F S.G SA W$B"

W.C = Do intero puntato: il punto aggiunge il 50% alla durata della nota.
HD = Re mezzo (durata metà di W)
QE = Mi quarto (durata metà di H)
I#F = Mi diesis ottavo (durata metà di Q)
S.G = Sol sedicesimo puntato (3/32 di W)
SA = La sedicesimo (2/32 di W, cioè metà di I)
W$B = Si bemolle ($B) intero

Per suonare quest'ultima serie di note, basterà:

PLAY "W.C HD QE I#F S.G SA W$B", oppure:
A$="W.C HD QE I#F S.G SA W$B":PLAY A$

Il "codice" è quindi molto simile a quello normale del pentagramma, solo che si definisce la durata prima della nota, mentre sul pentagramma è l'aspetto grafico della nota che ne definisce la durata.

```
ottave.list
0 play"o4"
5 print" OTTAVE";:
  color 4, 1: color 0, 16: color 6, 2
10 a$ = "cdefgab": print "o4 = default":
   play a$: print "Ottava:"
20 for t = 1 to 7: read o$
25 print o$
30 play o$ + a$
40 next t
100 data o0, o1, o2, o3, o4 ,o5, o6
```

Questo programma suona tutte le note intere (senza diesis o bemolle) delle 7 ottave del C-128. La riga 0 è una precauzione: se il calcolatore ha suonato in precedenza, l'ottava definita può essere diversa dalla quarta. PLAY "04" riporta al default. Il programma, dopo aver suonato le 7 note una volta nella quarta ottava, entra nel loop (20-40) e legge i 7 DATA, suonando la stringa nell'ottava definita da ciascuno mentre sullo schermo viene stampato il numero corrispondente. Notare come le stringhe vengono usate più o meno come per il PRINT. Per suonare due o più stringhe insieme è necessario usare il simbolo +. La lunghezza totale delle stringhe non deve superare i 255 byte. Una sola riga di programma ne può contenere solo 160 (4 righe dello schermo a 40, 2 righe di quello a 80): perciò è necessario usare una forma del tipo C$=A$+B$ per ottenere una stringa più lunga. Non è però consigliabile formulare stringhe lunghissime per il puro gusto di farlo: risultano meno leggibili ed è più facile fare confusione quando è necessario correggerle.
Ora che abbiamo sentito tutte le note, dal C dell'ottava o7 fino al B della o7, consideriamo più sistematicamente i codici da inserire nelle stringhe.

13.2 Durata delle note, diesis, bemolle

Oltre ai nomi delle note (C D E F G A B) ci sono:

W = nota intera (è il default)
H = metà di W
Q = un quarto di W
I = un ottavo di W
S = un sedicesimo di W
. (punto) = prolungare del 50% PLAY "I.C" dà a C una durata di 3/16.
= diesis
$ = bemolle

Come abbiamo già visto, la lettera che definisce la durata deve precedere la nota. WA significa "suonare La per una battuta intera", mentre HF significa suonare Fa per mezza battuta", e così via. Nel programma BACH, di cui si parlerà più avanti, abbiamo sempre una nota intera che suona mentre con un'altra voce suonano 4 note da un quarto. Provare a scrivere sulla tastiera:

```
PLAY "WC HD QE IF SG"
```

Ci sono altri due simboli importanti:

R è una nota che non suona — significa appunto "fare silenzio per la durata specificata". Quindi HR significa una pausa di mezza battuta.

M si usa quando suonano più voci simultaneamente. Indica che il calcolatore deve finire di suonare le note attuali prima di andare avanti.

Un'ottava, abbiamo detto, consiste delle note C-D-E-F-G-A-B-C, ossia Do-Re-Mi-Fa-Sol-La-Si-Do. L'ultima nota è ovviamente di nuovo Do (C), e perciò, se vogliamo essere precisi, dobbiamo dire che un'ottava consiste non di 8 note ma di 8 intervalli di una nota intera. Sulla tastiera del pianoforte ci sono tasti bianchi e neri: quelli bianchi danno le note intere mentre i neri danno i "diesis" e i "bemolle", che sono le note intermedie tra quelle intere: i neri vanno a gruppi di due e di tre. Do (C) è sempre il tasto bianco immediatamente a sinistra di un gruppo di due neri. Il Do al centro del pianoforte è appunto quello che si ottiene se si scrive PLAY "04 C". Poiché o4 è il default se non abbiamo suonato in precedenza, PLAY "C" basterà. Prendiamo ora la nota Re: sul pianoforte questa si trova tra ogni gruppo di 2 tasti neri: è il tasto bianco a destra del Do. Se premiamo il tasto bianco Re è come scrivere PLAY "D".
Premere il tasto nero a sinistra di Re: questo equivale a scrivere PLAY "$D", e la nota che viene prodotta è di altezza intermedia tra Do (C) e Re (D).
Premere il tasto nero a destra di Re: questo equivale a scrivere PLAY "#D", e la nota che viene prodotta e di altezza intermedia tra Re (D) e Mi (E).
Il programma Strumenti2 (si veda più avanti) suona esclusivamente le note nere del pianoforte. Per comodità sono tutte indicate con #, ma se si considera chiaramente come sono disposti i tasti del pianoforte, si capirà subito che suonare #C (Do diesis) è la stessa cosa di scrivere $D (Re bemolle); in entrambi i casi la nota prodotta è a metà strada tra C e D.
Per riassumere le indicazioni relative alla singola nota, si specificano: durata (W H Q I S .) e altezza (C D E F G A B # $). L'ordine è: durata seguita da altezza. Così

```
PLAY "H .$B"
```

suona la nota B bemolle (Si bemolle) per una durata di tre quarti.
Per rendere più leggibili le stringhe è possibile lasciare spazi senza che questi abbiano effetti sulle note suonate. Per scrivere stringhe lunghe bisogna però ricordare quel limite di 160 byte, che comprende gli eventuali spazi. Negli esempi di questo capitolo, ho seguito vari metodi. Forse quello usato per i DATA del programma BACH è il più chiaro.
Ma non abbiamo finito: esistono altri cinque comandi importanti da tenere in considerazione, anche se, dal momento che per ciascuno esiste un valore di default, possono essere omessi, almeno nella fase iniziale della creazione di una stringa musicale.

13.3 VOTUX

V =voce (da 1 a 3)
O =ottava (già discussa: da 0 a 6)
T =strumento (da 0 a 9)
U =volume (da 0 a 9)
X =filtro (1=attivo, 0=inattivo)

VOCE

Ci sono tre voci identiche che si possono usare indipendentemente e/o simultaneamente, in modo da ottenere accordi, anche usando strumenti diversi.
La voce default è V1. I due programmini Accordi1 e Accordi2 illustrano l'uso delle tre voci, mentre BACH ne dimostra forse più efficacemente le potenzialità.

PLAY "V1 C V2 E"

suona simultaneamente le due note Do e Mi: torneremo a questo argomento più avanti.

STRUMENTO

T indica lo strumento e corrisponde agli strumenti descritti con la specifica ENVELOPE. Il programma STRUMENTI1 dà un'idea sommaria delle possibilità e dell'uso.

```
strumenti1.list
5 print "STRUMENTI"; chr$(14):
  vol 10: color 4,1: color 0,16
10 a$ = "o4cdefgabo5.crr"
20 for t = 1 to 10: read t$, s$
25 print t$, s$
30 play t$ + a$
40 next t
100 data t0, pianoforte, t1, fisarmonica, t2,
        calliope, t3, batteria, t4, flauto, t5,
        chitarra, t6, clavicembalo, t7, organo, t8,
        tromba, t9, xilofono
```

Questo programma, oltre a illustrare i diversi suoni ottenuti con gli ENVELOPE di default, illustra come possiamo scrivere una frase musicale

(in questo caso la solita scaletta) e (con PLAY T$+A$) definire di volta in volta voci, ottave, strumenti, volumi o filtri diversi sommando stringhe diverse con +.

```
strumenti2.list
20 n = 60: tempo 80: color 4, 1: color 0, 16: color 5, 1:
   color 6, 16:vol10
30 trap 130
40 tempo 30:play"v104"
50 print " Premere un tasto ": getkey a$
60 read x$: x$ = "v1" + x$
70 a$ = "q#c q#d h#f #a #a q#c q#d h#f #a #a q#c q#d
         h#f #a #c h#aw #g q#a# g h#f #d h#d q#a #g
         h#f h#d h#d qa#g h#f #d #c# #dw #f"
80 print  x$; a$
90 play x$ + a$
100 goto 50
110 data o4 t0 u9, o6 t1 u7,o3 t2, o1 t3, o5 t4,
         o5 t5, o6 t6, o2 t7, o2 t8, o4 t9
120 return
130 if er = 13 then 200
```

Questo secondo programma prende un motivetto che viene suonato usando esclusivamente i tasti neri del pianoforte (tutte le note sono quindi con #). Ma per illustrare un po' meglio l'utilità dei vari strumenti varia anche l'ottava. Infatti non tutti gli strumenti sono fatti per suonare nella stessa ottava. Il pianoforte si sente bene nell'ottava o4, ma per esempio questa è un po' bassa per il flauto e un po' alta per illustrare il suono più tipico dell'organo. Poiché il programma presenta i numeri degli strumenti sullo schermo, è meglio seguirlo sullo schermo e con l'audio, che non su questa pagina. Gli strumenti sono:

0 = pianoforte
1 = fisarmonica
2 = calliope
3 = batteria
4 = flauto
5 = chitarra
6 = clavicembalo
7 = organo
8 = tromba
9 = xilofono

Si vedano il comando ENVELOPE e il programma def.env per i valori di default dei vari strumenti

Il calliope è una specie di organo a vapore che produce un suono un po' rauco ma non sgradevole.
Si veda più avanti anche il paragrafo su ENVELOPE.

VOTUX è un modo facile di ricordare i parametri che si possono specificare, ed è una buona idea specificarli sempre nello stesso ordine (anche se per il calcolatore non ha importanza).

VOLUME

U (volume) va da 0 a 9 (u0 è silenzioso, equivale a scrivere R invece di C D E F G A B). Vale per tutte le voci, per cui se si vogliono avere strumenti che suonano a volumi diversi in voci diverse è necessario specificare il volume ogni volta che si cambia voce (che è un po' noioso: sarebbe stato meglio se, come T per lo strumento, fosse indipendente per ciascuna voce). Il comando VOL per definire il volume globale della musica e U (indipendente per ciascuna voce) per il volume relativo. Ma si può sempre usare il comando del monitor o TV, che in fondo ha lo stesso effetto.

FILTER

X accende o spegne la funzione filtro. FILTER è un'istruzione usata per definire appunto il filtraggio dei suoni.
Un suono musicale non è mai un suono veramente "puro": rispetto a un tono puro prodotto da un oscillatore elettronico è quasi sempre un suono "sporco": comprende armoniche alte e basse, forme d'onda diverse fra loro, e così via. E la musicalità dipende in larga misura proprio da questa "sporcizia".
FILTER ci aiuta a controllare il tipo di "sporcizia" che desideriamo. La sintassi dell'istruzione è come segue:

FILTER *t, b, m, a, ris*

t frequenza di taglio: da 0 a 2047. È la frequenza di riferimento per *b, m* ed *a*.
b passa-basso: 1=attivo; 0=inattivo. Attenua tutte le frequenze superiori a *t*.
m passa-banda: 1=attivo: 0=inattivo. Lascia passare una banda di frequenze intorno al valore di *t*, attenuando le altre, superiori ed inferiori.
a passa-alto: 1=attivo: 0=inattivo. Attenua le frequenze al di sopra del valore di *t*.
ris determina la nitidezza del suono: il valore va da 0 a 15.

13.4 Durata delle note

Torniamo per un momento a questo argomento con un esempio. Come abbiamo già visto, le lettere S, I, E, Q, H, W definiscono la durata delle note (o meglio la durata relativa, perché TEMPO ne definisce quella assoluta). La lettera R è come una nota di "riposo": cioè non suona niente, ma può avere la stessa durata di una nota qualsiasi.
Questo programma usa IR, QR e R per definire riposi di un ottavo, di un quarto e di mezza nota.

```
sanmartino.list
5 print chr$(14); "SAN MARTINO"
10 vol 15: play "vl t7 u9"
20 f$ = " o5h cde.c irh cde.c qrh  ef.g qrh ef.g r qgagfhe.c ir
qgagfhe.c i.r we o4g #b ir o5we o4g #.b
25 print "Suono": print f$
30 play f$
```

13.5 ENVELOPE

Questa istruzione specifica stabilisce le precise caratteristiche di ciascuno degli strumenti che si suonano con la specifica PLAY (con T seguito da un numero da 0 a 9). Anche in questo caso, tentare una spiegazione completa richiederebbe molte pagine, e penso che il lettore capirà di più con la sperimentazione diretta. Basterà una riga di programma del tipo seguente:

```
30 envelope 0, 0, 9, 0, 0, 2, 512
```

La riga 30 ridefinisce T0 (pianoforte) come clavicembalo. Tenendo costante il primo zero, si possono variare gli altri valori entro i limiti indicati alla pagina 7-14 della System Guide, oppure nel Dizionario del BASIC in questo volume.
Teniamo sempre presente che non è assolutamente necessario variare l'ENVELOPE oppure usare filtri. Si riesce a suonare benissimo con i valori di default. Nonostante questo, sarà opportuno imparare a usarlo. Questo capitolo ne parla forse troppo poco, ma il Dizionario alla fine del volume ne spiega i concetti indispensabili. Data l'importanza dell'argomento si consiglia di usare il programma tutorial ENVELOPE nel Menu Musicale del dischetto: questo fornisce un panorama delle modificazioni dei suoni man mano che si variano i parametri, e ovviamente vale più di qualsiasi tentativo di spiegare a parole i suoni.

DEF.ENV

Il Menu Musicale prevede un metodo per restaurare gli ENVELOPE di default senza usare RUN/STOP-RESTORE. Quando viene caricato un programma musicale, il menu relativo provvede a registrare il fatto in due indirizzi di RAM assoluti al di fuori dell'area del BASIC. Quando il programma termina e ricarica automaticamente lo stesso menu, il contenuto di questi due indirizzi indica al menu di caricare def.env per restaurare gli inviluppi di default; def.env, a sua volta, ricarica il menu, che ora procede senza interruzioni. Il meccanismo è spiegato in maggior dettaglio in un altro capitolo.
Evidentemente, si potrebbe anche incorporare def.env in mus.menu. Nel caso del dischetto dimostrativo si voleva presentare un meccanismo utile anche per altri scopi. Per altre applicazioni il problema sarà spesso il contrario: quello di definire una nuova serie di strumenti, probabilmente senza più tornare ai default. Si può applicare la stessa procedura definendo ENVELOPE diversi. In questo caso, la serie di ENVELOPE originali potrebbe servire in moltissimi programmi diversi. È del tutto possibile ampliare il meccanismo usato e quindi far sì che def.env ricarichi in tutti i casi il programma che lo ha chiamato; questo può avvenire "silenziosamente", cioè senza che l'utente se ne accorga.

13.6 Tempo

Un'altra specifica semplice da usare, quasi indispensabile, è TEMPO. Questa non rientra nella stringa PLAY, ma è del tutto indipendente. TEMPO stabilisce la velocità della musica. La forma è

TEMPO *n*

dove *n* è un valore da 0 a 255. Il default è 8.

```
5 print"s"
30 for t = 1 to 255 step 5:tempo t: print "tempo =";t
40 :   a$ = "v0 o4 c d e f g a b o5 c"
50 :   play a$
60 next t
```

Queste righe daranno una (abbastanza divertente) illustrazione dell'effetto di TEMPO. Notare che il tempo per suonare la scala quando t=1 è tanto lungo da far temere che il calcolatore si sia guastato. TEMPO, ovviamente, cambia i valori della durata delle note. In altre parole se PLAY

"WF" è suonato a TEMPO 16, è come suonare PLAY "HF" a TEMPO 8. Questo fatto non è senza una certa utilità anche mentre si sta programmando:

- per accelerare un pezzo di musica che si sta provando, risparmiando tempo
- per rallentare un pezzo veloce, quando la stringa contiene un errore e il pezzo, con il tempo normale, va troppo veloce perché si possa sentire esattamente dov'è l'errore.

13.7 V

Anche se V è il primo parametro della serie VOTUX, l'ho lasciato all'ultimo perché finora abbiamo esaminato solo il modo di suonare una nota alla volta. Voce ci dà la possibilità di fare qualcosa di più. In ogni caso, anche se sembra una buona idea presentare i parametri nello stesso ordine, non è necessario seguire un ordine fisso.
Il C-128 ha tre voci. Ciò significa semplicemente che ha tre canali sonori separati. Ciò non significa che sia stereofonico: combina i tre canali insieme per l'uscita. Nonostante questa limitazione, il possesso di tre voci significa che si possono suonare tre note diverse contemporaneamente.
Il dischetto comprende 4 programmi per illustrare, a vari livelli di complessità, l'uso delle 3 voci. Sono: Accordi1, Accordi2, BACH1 e BACH2. Presentano delle note suonate con una voce sola, poi le combinano in due o tre voci per ottenere degli effetti particolari. Chi scrive, non essendo musicista, non ha potuto curare la bellezza musicale: i programmi danno soltanto esempi delle combinazioni.

```
bachl.list
10 trap420
20 color 0,16: color 5,1: color 6,16: print chr$(14): print"ssS"
30 print"   Johann Sebastian BACH"
40 a$ = "v2 o4 t0 qcdeg
50 b$ = "ffag"
60 c$ = "g o5c o4b o5c o4
70 d$ = "o4g e c d
80 e$ = "efga"
90 f$ = "gfed"
100 g$ = "eco3bo4c"
110 h$ = " do3gbo4d
120 i$ = "fede
130 tempo16
140 motivo$ = a$+b$+c$+d$+e$+f$+g$+h$+i$: print "Motivo": print
motivo$: play motivo$: sleep 2
```

```
150 accomp$ = "vl o3 t0 we f g g f e .d m   e f g e d qe hf wd c
160 print "Accompagnamento":print accomp$: play accomp$
170 print "Composizione"
180 envelope 1, 10, 10, 12, 9, 2, 2500
190 comp$ = ""
200 do
210 :  print comp$: play comp$
220 :  read comp$
230 loop until comp$ = "coda": comp$ = ""
240 data  vltl o3 w e  v2t7o4q cdeg
250 data m vl  o3 w f  v2 o4 q ffag
260 data m vl  o3 w g  v2 o4 q g  o5 c o4 b o5 c
270 data m vl  o3 w g  v2 o4 q gecd
280 data m vl  o3 w f  v2 o4 q efga
290 data m vl  o3 w e  v2 o4 q gfed
300 data m vl  o3 w d  v2 o4 q ec o3 b o4 c
310 data m vl  o3 h r  v2 o4 q d  o3 gb o4d
320 data m vl  o5 w e  v2 o4 q cdeg
330 data m vl  o5 w f  v2 o4 q ffag
340 data m vl  o5 w g  v2 o4 q g o5 c o4 b o5 c
350 data m vl  o5 w e  v2 o4 q gecd
360 data m vl  o5 w d  v2 o4 q efga
370 data m vl  o5 q e  v2 o4 q gf vl o5 h f   v2 o4 q ed
380 data m vl  o5 w d  v2 o4 q ec o3 b  o4 c
390 data m vl  o5 w c  v2 o4 q d  o3 gb o4 d
400 data coda
410 restore:goto170
```

Questa è una corale di BACH suonata con due voci, entrambe definite come "organo". Il pezzo ha il pregio di tornare sempre al punto di partenza. È doveroso precisare che l'ho semplificato notevolmente: resta comunque riconoscibile. L'organo della voce 2 ha l'ENVELOPE 7 di default, mentre ho ridefinito nella riga 180 l'inviluppo della fisarmonica per avere un accompagnamento organistico un po' diverso dall'organo principale.

```
180 envelope 1, 10, 10, 12, 9, 2, 2500
```

Questo ha un attacco (10) minore di quello della fisarmonica ma maggiore dell'organo di default, un decadimento (10) maggiore di entrambi, ha il sostentamento (12) della fisarmonica, un release alto (9, più di tutti i default elencati alla pag. 7-15 della System Guide). Ha naturalmente la forma d'onda 2 dell'organo (fisarmonica=1). L'ampiezza d'impulso è pure aumentata rispetto all'organo (2500 anziché 2048).
Ma per evitare che questo testo diventi troppo tecnico si suggerisce al lettore di compiere la seguente prova:

- caricare dal menu BACH1
- ascoltarlo, poi premere RUN/STOP e RESTORE e listare il programma
- inserire una REM all'inizio della riga 180 per renderla inattiva, oppure cancellarla
- dare il RUN e sentire la differenza quando la voce 1 suona con l'inviluppo della fisarmonica
- per riprendere il programma originale, selezionare Menu quando termina, poi ricaricarlo come prima.

Esaminiamo il programma dall'inizio.
Le righe 40-120 sono didattiche: in realtà non servono in questa forma. Dividono la musica in misure di quattro quarti (4/4), e rispecchiano il modo in cui ho creato la stringa globale, lavorando appunto con pezzettini di melodia. Ho lavorato sistematicamente, prima in modo diretto, scrivendo B$="FFAG" e così via. Quando PLAY B$ dava il risultato voluto, diventava una riga di programma. Quando finalmente tutta la stringa MOTIVO$ suonava bene, avrei dovuto scrivere, in modo diretto ma dopo aver dato il RUN alla parte di programma già completo

```
PRINT MOTIVO$
```

e poi inserire il numero 40 prima dell'output. Poi

```
DELETE 50-
```

avrebbe eliminato tutte le stringhette provvisorie. Le ho lasciate appunto per illustrare il metodo, e anche per sottolineare il fatto che, se avessi voluto, avrei potuto usare questi pezzi per comporre qualsiasi numero di stringhe complesse e diverse tra loro. Ancora, avrei potuto inserirle su tasti funzione (si veda l'esempio KEY2, più avanti). La stringa ACCOMP$ è invece stata ottenuta nel modo descritto, poi ho eliminato le righe provvisorie.
MOTIVO$ e ACCOMP$ suonano indipendentemente, per illustrare le due melodie da combinare insieme. Poi (e naturalmente il lavoro è più complesso) suonano simultaneamente.
Ho scelto la corale in questione perché:

- MOTIVO$ suona esattamente 4 volte più veloce di ACCOMP$: questo rende più chiara l'illustrazione, come si vedrà.
- Il pezzo è ciclico: cioè la prima nota può essere attaccata all'ultima, dando l'impressione di una COMP$ (composizione$) più lunga.

```
DATA
240 data v1t1 o3 w e v2t7o4q cdeg
250 data m v1 o3 w f v2 o4 q ffag
```

Prendendo queste due righe come esempi, notiamo che:

- Ogni riga di DATA corrisponde a una sola misura (cioè a 4 quarti). Nulla ci impedisce di scrivere in una stessa riga il contenuto di più stringhe (una riga di DATA può contenere fino a 160 caratteri), ma questo sistema appare più chiaro.
- La nota suonata da V1 (W=intera) corrisponde quindi a 4 note suonate da V2.
- M all'inizio di ogni DATA fa attendere che le note dell'ultima misura (o battuta) siano tutte terminate prima di proseguire (il lettore ne elimini qualcuna e poi ascolti il programma).
- Per le due voci usate (V1 e V2) si specifica una sola volta lo strumento (T1 e T7). T1, come abbiamo visto, è stato ridefinito con ENVELOPE. L'ottava (o3 e o4 nelle due righe sopra) va definita ogni volta che si cambia: poiché in questo pezzo le ottave sono sempre diverse per le due voci, si specifica sempre.
- La stessa osservazione vale per la durata (W o Q in questi esempi) della nota.
- L'ultimo DATA è "coda", che non viene suonato, ma è usato come segnale per finire il loop.
- Come già osservato, la spaziatura non ha effetto sul suono né nella stringa né (come si vede) nei DATA.

Una seconda versione del programma BACH aggiunge la V3, che fornisce un basso continuo. Ora abbiamo due ENVELOPE fatti su misura: quello nuovo è definito oboe. A parte lo spostamento delle due definizioni di ENVELOPE (righe 5-6) la struttura del programma è la stessa. Si aggiunge la V3 e definizioni di volume diverse per i tre strumenti.
Prendendo questo programma come base può essere ancora una buona idea variare alcuni parametri per capire le differenze che si creano. Entrambi i programmi BACH sono strutturati in modo che sia necessario premere RUN/STOP e RESTORE per uscire dall'alternanza programma/menu (grazie al doppio trap; infatti la 420 contiene TRAP 420). Ciò serve per partire sempre dagli ENVELOPE di default, ma è anche un metodo per dare un po' di protezione a un programma.
Il compositore, non importa se usi il calcolatore o la più tradizionale carta da musica, deve essere metodico. Usare i DATA è in genere la soluzione più efficace (è anche facile duplicare le righe di DATA, e ciò è utile anche se richiedono lievi modifiche). Molte delle righe DATA in questo programma sono state duplicate e un po' modificate: il metodo è più veloce e garantisce una maggiore precisione.
È più facile essere metodici usando i DATA. Si può scrivere, usando una riga per ciascun DATA, tutta la parte di una voce (V1 nel nostro caso), poi, quando suona perfettamente, aggiungere più a destra in ciascuna ri-

ga, la seconda voce, e così via. Sarà ovviamente necessario modificare il READ relativo.
Notare che, in ogni misura o battuta, bisogna precisare prima la nota o le note che devono suonare più a lungo, nell'ordine W H Q I S: se non si fa così si perde la sincronizzazione della musica. Ciò, usato con conoscenza di causa, può essere utile per sincopare il ritmo, ma in un pezzo formale come il nostro, sarebbe un disastro. Provare per capire: basta modificare un paio di DATA.
La ripetitività del lavoro può essere ridotta in vari modi. Particolarmente utile è usare i tasti funzione:

```
key1,"v3 t0 u5"
key2,accomp$
key3,"comp$+"
```

... e così via, possono far risparmiare tempo e fatica e ridurre il rischio di errori. Basta ovviamente premere il tasto funzione voluto per scrivere anche una stringa lunga. Key2, nell'esempio sopra, contiene non i caratteri "ACCOMP$", ma la stringa musicale stessa. In altre parole, la pressione di F2 non stamperà la stringa "accomp$", ma

```
v1 o3 t0 we f g g f e .d m e f g e o d qe hf wd c
```

Se dopo avere creato il motivo principale esiste la necessità di inserire molte volte uno stesso gruppo di caratteri in mezzo ai gruppi di data già scritti, si può premere ESC-A (auto-inserimento), portare il cursore al punto voluto e premere il tasto funzione: si aprirà lo spazio necessario per inserire la stringa. Si riduce così il rischio di errore.

13.8 SOUND

Si è volutamente trascurato il comando SOUND. Questo è molto potente, più adatto agli effetti sonori che alle composizioni musicali. La voce nel Dizionario del BASIC ne fornisce un resoconto, e il Menu Musicale del dischetto che accompagna il libro comprende il programma SOUND che costituisce in effetti un tutorial sulle possibilità fornite dal comando, anche se non può illustrare tutte le combinazioni possibili, che sono molte migliaia. È un caso in cui la sperimentazione personale può dare, dopo una certa confusione iniziale, più aiuto che una descrizione in queste pagine. I commenti che il programma stampa sullo schermo, poiché sono accompagnati dai suoni realmente prodotti, sono probabilmente più eloquenti che molte pagine di testo.

14

Dizionario del BASIC 7.0

Questo dizionario è organizzato in un'unica sezione in ordine alfabetico. Ogni voce inizia con un'intestazione che fornisce le seguenti informazioni:

PAROLABASIC **Comando / Istruzione / Funzione...** **Versioni**

Sintassi:
PAROLABASIC [*parametro* [,*parametro*]]

Abbreviazione:

Breve spiegazione

Comando, Istruzione, Funzione, Variabile del sistema. Nel resto del libro non è sembrato utile, in molti casi, dare particolare peso alla distinzione, ormai quasi teorica, tra comando e istruzione. È spesso stato usato il termine "comando" o semplicemente "voce". In questa sezione la distinzione è mantenuta.

Versioni: si indicano i calcolatori Commodore che hanno una voce identica o molto simile. L'indicazione **16** si riferisce sia al C-16 che al Plus/4.

Sintassi: in genere viene seguita la notazione standard usata: il maiuscolo indica parole chiave e comandi che devono essere digitati come sono scritti; il corsivo indica i parametri che possono assumere diversi valori. Le parentesi tonde devono essere digitate in quanto fanno parte della sintassi; le parentesi quadre racchiudono i parametri opzionali. La Commodore propone una sintassi alternativa (con la parola ON) per i comandi del disco: non è stata riportata perché di scarsa utilità. In

certi casi in cui le parentesi quadre sono numerose, si è preferito scrivere più righe di formati alternativi. Si noti che in molte voci (si veda per esempio CIRCLE), i parametri specificati sono separati da virgole. Se questi parametri sono tra parentesi quadre possono essere omessi: ma se si desidera specificare un parametro successivo è necessario inserire una virgola per ogni parametro precedente, anche se non si desidera cambiarne il valore:

CIRCLE,10,10,10,,,,,90

Abbreviazione: sono riportate tutte le abbreviazioni, anche quelle che risparmiano un solo byte. In linea di massima, molte abbreviazioni del BASIC 7.0 non sono molto utili: se una voce non si usa spesso è meno faticoso scriverla per esteso che ricordarne l'abbreviazione. Se però si deve per esempio programmare i tasti funzione per qualche lavoro speciale (nel quale caso il numero dei byte occupati può essere importante), ogni abbreviazione è utile, perché occupa meno memoria nella stringa riservata al tasto funzione. All'interno di un programma, invece, l'interprete espande le voci abbreviate, per cui le abbreviazioni servono solo per risparmiare tempo mentre si scrive. Le abbreviazioni vanno battute esattamente come nel testo (premendo cioè SHIFT per ottenere le maiuscole).

La spiegazione della voce tenta di rendere chiaro non solo l'uso ma anche l'utilità dell'operazione. Nel caso di certe voci trattate più in particolare nei capitoli precedenti di questo libro, per evidenti motivi di spazio, la spiegazione qui è succinta, e si rimanda al capitolo relativo.

ABS **Funzione** **20, 16, 64, 128**

Sintassi:
v=ABS(*x*)

Abbreviazione:
aB

Dà il valore assoluto di un numero o di una variabile numerica.
In pratica il valore assoluto di un numero è sempre un numero positivo oppure 0.

AND **Operatore logico** **20, 16, 64, 128**

Sintassi:
IF *a*=1 AND *b*=2 THEN...
oppure: *a*=*x* AND *y*

La prima significa "se a=1 e b=2"; cioè se entrambe le espressioni sono vere... Questo uso, come operatore relazionale, è il più frequente per chi usa il BASIC, particolarmente il BASIC 7.0, che elimina molte occasioni in cui era necessario usarlo anche per creare una maschera per i bit di un valore. Un caso in cui è ancora necessario è con la funzione BUMP, dove un valore numerico unico contiene, rispettivamente per gli sprite da 1 a 8, un 1 per gli sprite che hanno subito collisioni, e uno 0 per gli altri. Così per esempio un valore di 2 restituito da PRINT BUMP(1) o da PRINT BUMP(2) corrisponde al numero binario 00000010 e indica che lo sprite 2 (bit 2) ha avuto una collisione.
La tabella della verità per AND è:

```
0 AND 0    1
1 AND 0    0
0 AND 1    0
1 AND 1    1
```

Si vedano anche le voci OR, NOT, XOR. Programma ORANDXOR.

APPEND **Comando** **128**

Sintassi:
APPEND #*nf*, "*nomefile*" [,D*n*] [,U*n*]

Abbreviazione:
aP

Appende, cioè aggiunge in coda, dati a un file sequenziale già esistente sul disco.

```
APPEND #1, "file di ieri"
PRINT #1, "Questo lo aggiungo oggi"
CLOSE 1
```

"*nomefile*" può essere contenuto in una variabile, la quale deve essere tra parentesi:

```
A$="NOMEFILE"
APPEND #1, (A$)
```

ASC **Funzione** **20, 16, 64, 128**

Sintassi:
v=ASC(X$)

Abbreviazione:
aS

PRINT ASC("A") dà il numero di codice ASCII della lettera A.
PRINT ASC(X$) dà il numero di codice ASCII del *primo* carattere della stringa X$. Ignora i caratteri successivi.

ATN **Funzione** **20, 16, 64, 128**

Sintassi:
v=ATN (x)

Abbreviazione:
aT

v assume il valore (in radianti) dell'angolo avente tangente x.

AUTO **Comando** **16, 128**

Sintassi:
AUTO [n]

Abbreviazione:
aU

Modo diretto. Numera automaticamente le righe di un programma BASIC. L'argomento n dà l'intervallo tra i numeri.

1 Impostare AUTO 10 (per esempio)
2 Compilare una riga con un numero n qualsiasi
3 Battere RETURN.

All'inizio della nuova riga appare il numero successivo (n+10). Questo ac-

cadrà tutte le volte che si compilerà una nuova riga. Per disattivare AUTO, basta scrivere AUTO senza argomento e battere RETURN.

BACKUP — Comando — 128

Sintassi:
BACKUP D*n* TO D*n* [,U*x*]

Abbreviazione:
baC

Solo per drive doppio. Permette di copiare un intero dischetto, compresa la formattazione, in una sola operazione. All'interno del drive doppio ci sono due dischetti. Uno si trova nel drive 0, l'altro nel drive 1. Per copiare dal drive 0 a un nuovo dischetto nel drive 1:

```
BACKUP D0 TO D1
```

Se invece il drive doppio non è l'unico drive (unità), specificare anche il numero dell'unità:

```
BACKUP D0 TO D1 ON U9
BACKUP D0 TO D1, U9
```

daranno entrambi lo stesso risultato.

BANK — Istruzione — 128

Sintassi:
BANK *n*

Abbreviazione:
bA

Il C-128 ha fino a 16 banchi di memoria: quando si usa PEEK, POKE, e SYS, per accedere a un banco diverso dal 15 bisogna cambiare il numero di BANK prima di procedere. I banchi 2, 3, 6, 7, 10, 11 non esistono: sono tenuti liberi per espansioni di memoria. Si vedano anche i comandi BLOAD e BSAVE. Per i vari banchi (numerati 0-15) si veda il capitolo sulle espansioni di memoria.

BEGIN... BEND **Istruzione** **128**

Sintassi:
IF (*espressione*) THEN BEGIN [*routine*] BEND

Abbreviazione:
bE, beN

Dà la possibilità di avere un'istruzione condizionale, cioè una routine anche lunga, estesa su più righe, senza dover fare precedere a ogni riga la stessa clausola IF.

Esempio:

```
5  input x
10 if x = 1 then begin
15 :    print "visto che x = 1"
20 :    print "stampo queste due righe"
25 bend: else begin......
```

È come se le righe 15 e 20 cominciassero anch'esse con "IF X=1". La istruzione BEGIN, che come ELSE si aggiunge alla famiglia IF del C-128, è trattata nel Capitolo 4. Notare la posizione di ELSE (se richiesto). Si vedano anche la voce IF e il programma BEGIN/BEND (Lato 1 del disco).

BLOAD **Comando** **128**

Sintassi:
BLOAD "*nomefile*" [,D*n*] [,U*n*] [B*n*] [,P*n*]

Abbreviazione:
bL

Carica un file precedentemente salvato in codice binario dal drive *n* sull'unità U*n*, inserendo i dati nel banco B*n* a partire dall'indirizzo P*n*. La System Guide descrive questo comando solo per salvare file di sprite. Questo libro ne discute inoltre l'uso per salvare schermi di testo e schermi grafici.
Se non si specifica P, il file viene caricato all'indirizzo dal quale era stato salvato. Si veda anche BSAVE. Nel direttorio i file binari appaiono come "prg" (programmi). Si suggerisce di usare un prefisso.

Esempio:

BLOAD "sp.freccia", B0, P3584

BOOT **Comando** **20, 128**

Sintassi:
BOOT "*file*", [D*n*] [,U*n*]

Abbreviazione:
bO

BOOT senza argomento serve solo su un disco "boot". Il disco che accompagna questo volume è un disco boot. Accendere il drive, inserire il disco, poi accendere il computer. Il 128 carica il "boot program" e lo esegue. Questo è un programma breve che dice al calcolatore di cercare un altro programma (in BASIC o in linguaggio macchina) e di eseguirlo. L'altro programma può essere a sua volta un programmino del tipo:

100 print "Attendere, prego": run "programmone"

Questa soluzione è comoda se si desidera poter cambiare il nome del programma da eseguire. Sul nostro disco, BOOT carica un menu.
Il C-128 DOS SHELL contiene il programma AUTOBOOT MAKER, che inserisce la routine che carica il "boot program". La procedura, già descritta nel primo capitolo di questo volume, è semplice. Non è necessario che "programmone" esista prima di eseguirla, ma naturalmente dovrà essere messo sul dischetto in un secondo tempo. Se la traccia del disco usata da BOOT contiene già un file, possono insorgere problemi: il file, ovviamente, verrà perduto (ma AUTOBOOT MAKER avverte del rischio). Senz'altro la soluzione migliore è di usare un dischetto nuovo, appena formattato, poi copiare su questo i file desiderati, facendo attenzione a lasciare sempre un certo numero di blocchi liberi. Una buona regola può essere: lasciare almeno tanti blocchi liberi quanti ne occupa il file più lungo del disco, o almeno quanti ne occupa l'ultimo file da salvare. Se quindi restano 50 blocchi liberi, dovrebbe essere possibile ancora salvare senza rischi un file di 25 blocchi.
Il programma AUTOBOOT MAKER, anche se può funzionare solo con il C-128 (a differenza di altri programmi sul disco), non fa parte del "C-128 DOS Shell": si carica indipendentemente. È in BASIC.

BOX **Funzione** **16, 128**

Sintassi:
BOX [*fonte colore*],*x1,y1* [,*x2,y2*] [,*angolo*] [,*paint*]

Abbreviazione:
nessuna

Disegna un rettangolo tra le coordinate *x1, y1* [,*x2,y2*]

fonte colore=il colore della linea che delimita il rettangolo: 0=colore dello sfondo: 1=colore del carattere; 2=multicolor 1; 3=multicolor 2. (2 e 3 funzionano solo con GRAPHIC 3 e GRAPHIC 4).
x1 e *y1* sono le coordinate dell'angolo superiore sinistro. I valori dipendono dalla scala usata (si veda SCALE).
x2 e *y2* sono le coordinate dell'angolo inferiore destro. Se non specificate assumono il valore della posizione del pixel cursor.
angolo: ruota il rettangolo in senso orario del numero di gradi specificato.
paint: se *paint*=1, si riempie dello stesso colore specificato all'inizio. Se *paint*=0 oppure è omesso, non accade niente.

Esempi:

```
BOX 1, 20, 20, 70, 70
```

disegna un rettangolo tra *x*=20, *y*=20 ed *x*=70, *y*=70. Il colore 1 sarà stato precedentemente determinato con COLOR1,*n* (dove *n*>0 <17).

```
BOX 1, 20, 2, 70, 70, 45, 1
```

disegna lo stesso rettangolo, ma lo ruota di 45 gradi e lo riempie del colore indicato.
Tranne *x1, y1,* qualsiasi parametro può essere omesso, ma la "sua" virgola deve rimanere.
TRAPPOLA: se si omettono *x2* e *y2* lasciate al loro posto una sola virgola.
Sembra curioso, ma può essere utile, il fatto che i gradi di rotazione possono essere più di 360. 360=0: così 370 gradi=10 gradi...
Si veda anche il breve programma BOX.

BSAVE **Comando** **128**

Sintassi:
BSAVE "*nomefile*" [,D*n*],[U*n*] [,B*n*] P*n1* TO P*n2*

Abbreviazione:
bS

Va a leggere gli indirizzi tra P*n1* e P*n2*, e ne salva il contenuto su disco in forma binaria. Come al solito D è seguito dal numero di drive, U dall'unità (default 0 e 8 rispettivamente). Si vedano la voce BLOAD e i Capitoli 2 e 6 di questo libro. La System Guide fornisce pochi ragguagli.
Il Lato 2 del dischetto comprende due programmi, SALVAGRAFICI e SALVASPRITE, che consentono di salvare e/o di caricare file rispettivamente di grafici e di sprite. Una grave lacuna nella System Guide è l'assenza di un'indicazione relativa al fatto che gli sprite non devono assolutamente essere accesi mentre vengono salvati. Ciò provoca infatti effetti imprevedibili: tra le varie possibilità può succedere che il programmino BOOT nella traccia 1, settore 0, del disco venga danneggiato, oppure che, al momento di ricaricare gli sprite, avvengano scritture assurde nella zona di memoria immediatamente successiva all'area degli sprite: vengono messi fuori azione HELP e i tasti funzione.
A chi non desiderasse usare SALVASPRITE, si suggerisce di usare sempre una routine come:

```
FOR T=1 TO 8: SPRITE T, 0: NEXT
BSAVE"sp.[nome]",B0, P3584 TO P4096
```

BUMP **Funzione** **128**

Sintassi:
BUMP(1) oppure BUMP(2)

Abbreviazione:
bU

Indica quali sprite hanno subito collisioni dopo l'ultimo controllo. BUMP(1) indica gli sprite che hanno urtato altri sprite: BUMP(2) indica quali sprite hanno urtato altri oggetti sullo schermo. I valori si azzerano dopo che BUMP è stato usato.
BUMP non dà un valore immediatamente comprensibile. I dati per tutti gli otto sprite sono contenuti in un solo byte, cioè ogni bit corrisponde ad uno sprite. Così, come spiega la System Guide, se BUMP(*n*) dà il valore 16, indica che solo lo sprite 4 ha avuto una collisione, poiché $16=2^4$.
In altre parole, il valore dato da BUMP è un byte che rappresenta un valore decimale 0-255.

Esempio:

PRINT BUMP (1)
144

Il valore 144 restituito da BUMP corrisponde al numero in codice binario:

10010000

Leggendo da destra a sinistra, significa che gli sprite 5 e 8 (i due 1) sono entrati in collisione con altri sprite. L'istruzione:

FOR T=0 to 7: IF BUMP(1) AND 2^T THEN T=T+1 : NEXT : ELSE NEXT

fornirà un elenco degli sprite in questione. I bit, infatti, si numerano da 0 a 7, da destra verso sinistra. Si veda la voce AND.

CATALOG — Comando — 128

Sintassi:
CATALOG [D*n*] [U*n*] [,*stringa di ricerca*]

Abbreviazione:
cA

Si veda DIRECTORY. Non è spiegato perché si è introdotto un comando duplicato. CATALOG e DIRECTORY danno esattamente lo stesso risultato. Unico vantaggio (piccolissimo): l'abbreviazione è cA (C SHIFT-A), che è più breve di diR.

CHAR — Istruzione — 16, 128

Sintassi:
CHAR [*fonte colore*],*x*,*y*[,*stringa*] [,*rvs*]

Abbreviazione:
chA

Stampa un carattere o una stringa nella colonna *x*, riga *y* dello schermo (sia lo schermo grafico che quelli normali a 80 e a 40 colonne). Già di-

scussa in questo libro, è un'istruzione utilissima, sia per la grafica (per la quale è indispensabile), sia sugli schermi normali. *fonte colore* è, come in altri casi: 0=sfondo; 1=colore principale; 2=multicolor 1; 3=multicolor 2. I valori *x* e *y* non sono influenzati da SCALE, quindi sono sempre tra 0 e 79 e tra 0 e 24, rispettivamente per *x* e *y*. Sullo schermo a 40 colonne, quindi, un numero >39 per *x* andrà sulla riga successiva: è consigliabile quindi non superare 39 (per evitare confusioni), a meno che l'istruzione non debba essere usata anche sullo schermo a 80 colonne.
Sugli schermi normali funziona come un PRINT e i colori, il reverse, ecc. si ottengono allo stesso modo. In modo grafico, *rvs* può essere 1 o 0: 1 stampa la stringa in campo inverso. Provare:

```
10 char,10,10,str$(ti)+" "+ti$
20 goto10
```

Ottenere lo stesso effetto con PRINT è molto difficile! Serve bene anche per comporre tabelle sullo schermo, anche se PRINT USING è spesso più efficace.
L'istruzione CHAR è discussa nei Capitoli 5 e 8 di questo libro, e il dischetto che accompagna il volume ne contiene varie applicazioni, oltre al tutorial omonimo.

CHR$ — Funzione — 20, 16, 64, 128

Sintassi:
X$=CHR$(*n*) oppure PRINT CHR$(*n*)

Abbreviazione:
cH

Restituisce o assegna il carattere corrispondente a un determinato codice ASCII.

```
PRINT CHR$(65)
A
X$=CHR$(65): PRINT X$
A
```

Entrambi fanno apparire una A sullo schermo. Esistono certi caratteri "non stampabili" come CHR$(13) [RETURN] o CHR$(27) [il valore ESCAPE, indispensabile con la stampante]: CHR$ può essere l'unico modo di trasmettere al calcolatore o alla periferica questi codici.

```
10 for t = 0 to 255
20 :  print t; "="; chr$(t);
30 next t
50 open 4, 4
60 for t = 0 to 255
70 :  print#4, t; "="; chr$(t)
80 next t
90 close 4
```

Stampa prima sullo schermo e poi sulla stampante il carattere corrispondente a CHR$(t). Sulla stampante avranno luogo anche operazioni (come avanzamento modulo) che non hanno alcun effetto sullo schermo. Si veda i programmi ASCII TABELLA e ASCII CBM sul disco.

CIRCLE **Istruzione** **16, 128**

Sintassi:
CIRCLE [*fonte colore*], *x,y,xr* [*,yr*] [*,iniz*] [*,fine*] [*,rot*] [*,inc*]

Abbreviazione:
cI

Disegna un cerchio, ellisse, arco.

fonte colore: si veda CHAR, sopra.
fine coordinate (influenzate da SCALE)
xr raggio di un cerchio, raggio *x* di un'ellisse (SCALE)
yr raggio verticale di un'ellisse (SCALE)
iniz angolo d'inizio di un arco (default=verticale=0 gradi)
fine angolo di fine di un arco (default 2 gradi)
rot rotazione in gradi (senso orario, default 0)
inc gradi tra un segmento e un altro (default 2 gradi)

Come sempre, i parametri in parentesi quadre sono opzionali, ma la virgola deve essere inserita, come negli esempi:

CIRCLE1,160 ,100 ,50 disegna un cerchio.
CIRCLE, 160 ,100 ,50, 20 disegna un'ellisse.

Si possono ottenere anche poligoni, per esempio:

CIRCLE,60, 140 ,20, 18 ,,,,120 disegna un triangolo.

Questa istruzione è discussa nel Capitolo 8 di questo volume, ed è illustrata dai programmi CIRCLE e INC (accessibile dal menu finale di CIRCLE), nonché nei programmi del gruppo DRAW sul Lato 2 del dischetto.

CLOSE — Istruzione — 20, 16, 64, 128

Sintassi:
CLOSE *n*

Abbreviazione:
clO

Chiude un file su nastro, disco, stampante, ecc.

CLOSE 7

chiude il file #7 non importa su quale periferica.
La mancata chiusura di un file su disco comporta la perdita dei dati o di una parte di essi: nel direttorio un file non chiuso (generalmente un file sequenziale; i file "prg" vengono chiusi automaticamente) è evidenziato con un asterisco. Si veda anche la voce COLLECT. Sulla stampante CLOSE è necessario in molti casi per ritornare al normale funzionamento del calcolatore. Si veda anche la voce CMD. Si vedano anche le voci OPEN, DOPEN, DCLOSE.

CLR — Istruzione — 20, 16, 64, 128

Sintassi:
CLR

Abbreviazione:
cL

Libera tutta la memoria (banco 1) occupata da variabili senza toccare il programma in BASIC (banco 0). Si veda anche NEW.

CMD — Comando — 20, 16, 64, 128

Sintassi:
CMD *n* [,*stringa*]

Abbreviazione:
cM

L'uso principale di questo comando è in modo diretto. Serve in particolare per stampare il listato di un programma.

```
OPEN 4,4
CMD 4, "PROG.1 LIST"
LIST
PRINT#4
CLOSE 4
```

Queste righe (da programma o da tastiera, ma è consigliabile usare CMD da tastiera) aprono il file #4 per la stampante (il numero # del file può essere da 1 a 255, mentre il numero della stampante deve essere 4: vige però l'abitudine di numerare il file come 4 se l'indirizzo secondario è 0, oppure come 7 se l'indirizzo secondario è 7). CMD 4 ordina alla stampante di scrivere l'intestazione "Prog.1 list". LIST elenca le righe del programma PROG.1, che non appaiono sul video ma sulla stampante. PRINT#4 ridirige l'output verso lo schermo (importante per non perdere il controllo del calcolatore); infine CLOSE 4 chiude la linea con la stampante.

COLLECT **Comando** **16, 128**

Sintassi:
COLLECT [D*n*] [,U*n*]

Abbreviazione:
collE

Riordina un dischetto. Quando un'operazione di scrittura su disco non è andata a buon fine (di regola perché il programmatore si è scordato di chiudere il file) resta un file vuoto, che però occupa spazio. Appare nel direttorio con un asterisco:

```
0    PASTICCIACCIO    *SEQ
```

Se abbiamo appena commesso l'errore, è probabile che un CLOSE oppure un DCLOSE metta in ordine il disco e che il file si riempia di dati, perché se questi sono presenti in memoria, il calcolatore resta infatti in attesa del CLOSE per portare a termine le operazioni di scrittura.
Se questo non è possibile (in altre parole, se il calcolatore ha nel frattempo eseguito altre operazioni che hanno eliminato il buffer contenente i dati destinati al file) non usare SCRATCH: questo è un file anomalo, e cercare di farlo può anche danneggiare il direttorio. COLLECT è l'unica (e peraltro comoda) soluzione.
Se un dischetto è importante, vale la pena di controllare se il totale dei blocchi assegnati a ciascun file (nella colonna a sinistra del direttorio)

corrisponde al totale BLOCKS FREE (Blocchi Liberi) in fondo al direttorio: il programma DIRTUT/DISCOTUT calcola la somma dei blocchi liberi e di quelli occupati e se il totale è diverso da 640 o da 1328 (rispettivamente per il 1541 e per il 1571), segnala il fatto. Notare che una differenza di 1 può essere dovuta alla presenza di una routine di BOOT, non indicata nel direttorio. DIRTUT non segnala questo soltanto se il nome del disco contiene la parola "boot" (i nostri dischi hanno i nomi LATO1-BOOT e LATO2-BOOT).
Se c'è una differenza, COLLECT dovrebbe mettere ordine: infatti se dopo COLLECT (che impiega anche più di un minuto se il disco è abbastanza pieno) c'è un totale di BLOCKS FREE diverso dal totale precedente, abbiamo la conferma che c'era dello spazio assegnato male. Per scrupolo potremmo trasferire i file, a uno a uno, su un dischetto nuovo, come suggerisce il manuale del drive 1571.
COLLECT non deve essere usato se sul dischetto esistono file random (ad accesso diretto): essi operano in modo diverso e perderebbero spazio (e dati). Questa raccomandazione vale per esempio per molti database. DIRTUT si rifiuterà di eseguire COLLECT se sono presenti file con le sigle REL e USR.
Inoltre, se per sostituire un file si è usato DSAVE "*@nomefile*" al posto di SCRATCH seguito da DSAVE, può essersi verificato un gravissimo errore, spesso irreparabile. Per un errore di scrittura nel direttorio due file si sono sovrapposti e, entrando in uno dei due, si trova a un certo punto che il drive ha letto anche una parte dell'altro. Per un utente non esperto esistono poche possibilità di salvare uno dei due. DIRTUT in questo caso segnala un totale di blocchi (liberi+occupati) superiore a 640 o a 1328. Non eseguire COLLECT prima di aver fatto una copia. Controllare il disco fino a scoprire quale file è anomalo (spesso i due file coinvolti avranno lo stesso numero di blocchi). Se il file anomalo non è importante, eliminarlo con SCRATCH. Se invece il file anomalo è quello importante vale la pena (senza molte speranze) di usare SCRATCH sull'altro. COLLECT, in ogni caso, renderà permanente l'anomalia. Usare la copia del disco per i tentativi: se il file danneggiato è prezioso si potrà sempre portare l'originale da un esperto.
Per riassumere: con un drive Commodore è necessario un atteggiamento rassegnato. Neppure il 1571 è molto efficiente.

COLLISION **Istruzione** 128

Sintassi:
COLLISION *tipo* [,*specifica*]

Abbreviazione:
colL

L'istruzione serve a definire il comportamento del calcolatore in caso appunto di collisione tra sprite e sprite (*tipo* 1), tra sprite e altri oggetti sullo schermo (*tipo* 2) o tra sprite e penna luminosa (*tipo* 3).
È in pratica una specie da GOSUB: per sapere che collisioni hanno luogo si usa la funzione BUMP.

COLOR **Istruzione** **16, 128**

Sintassi:
COLOR *fonte, colore*

Abbreviazione:
colO

Esistono 7 "aree" per il colore sullo schermo: *fonte* va quindi da 0 a 6.

0 = sfondo per lo schermo a 40 colonne
1 = colore di primo piano (40 colonne)
2 = multicolor 1
3 = multicolor 2
4 = bordo (40 colonne)
5 = colore del carattere (40 e 80)
6 = sfondo per lo schermo a 80 colonne

Il valore di *colore* è un valore da 1 a 16

1 = nero	2 = bianco
3 = rosso [r. scuro]	4 = cyan [c. chiaro]
5 = viola [v. chiaro]	6 = verde [ve. chiaro]
7 = blu [b. scuro]	8 = giallo [g. chiaro]
9 = arancione [v. scuro]	10 = marrone
11 = rosso chiaro	12 = grigio scuro [cyan scuro]
13 = grigio medio	14 = verde chiaro
15 = blu chiaro	16 = grigio chiaro

Le indicazioni tra parentesi quadre [ ] sono i colori (qualche volta diversi, come si vede) che si ottengono sullo schermo a 80 colonne.

COLOR 4,1 colora il bordo dello schermo a 40 colonne in nero.
COLOR 5,3 colora i caratteri (40 o 80 colonne) in rosso (normale o scuro rispettivamente).

COLOR 6, 16 oppure COLOR 0, 16 colorano rispettivamente lo schermo a 80 e a 40 di grigio chiaro. Questo sembra il colore migliore quando si devono leggere testi sullo schermo: la scelta ovvia (2=bianco) è troppo abbagliante sullo schermo a 40 colonne.

È deludente il fatto che con il BASIC 7.0 non sia possibile variare la luminosità del colore. Questo era uno dei pochi pregi della serie C-16/Plus/4.

CONCAT **Comando** 128

Sintassi:
CONCAT "*file2*" [,D*n*] TO "*file1*" [,D*n*] [,U*n*]

Abbreviazione:
cO

Concatena due file, cioè attacca un file alla coda di un altro. Se non si indicano D e U, questi, come al solito, hanno i valori di default 0 e 8.
Il nuovo file conserva il nome "*file1*": è meglio usare nomi di file con 10 caratteri o meno: l'intero comando (a partire dalla prima C di CONCAT non deve superare i 41 caratteri. Infine (e questo è utile se il comando è usato in un programma), si possono usare variabili tra parentesi come nel secondo esempio qui di seguito:

Esempi:

```
10 concat "coda" to "testa"

10 a$="testa": b$="coda"
20 concat (b$) to (a$), U9
```

Entrambi gli esempi creano un file "testacoda" con il nome "testa": il secondo usa l'unità 9. Si veda il programma DIRTUT/DISCOTUT.
L'efficienza di questo comando non è perfetta. Si possono avere, infatti, dei problemi al punto della "cucitura" tra i file, poiché non viene correttamente eliminata la stringa di chiusura del file "testa". Si può usare il word processor per correggere l'errore, ma un word processor normale è comunque capace di concatenare due file senza errori, il DOS della Commodore no.

CONT **Comando** **20, 16, 64, 128**

Sintassi:
CONT

Abbreviazione:
nessuna

Riprende l'esecuzione di un programma dopo una interruzione, se questa non è stata provocata da una condizione d'errore. Uno STOP (sia dal tasto STOP che dall'istruzione BASIC) oppure l'istruzione END possono essere revocati con CONT, purché non esista una condizione d'errore e non ne vengano provocate durante il periodo in cui il programma è fermo. Oltre alla condizione d'errore, il programma non riprenderà se è stata fatta qualche modifica al programma (come l'inserzione di una nuova riga o la correzione di una riga esistente). In questi casi si riceve il messaggio CAN'T CONTINUE ERROR.

COPY **Comando** **16, 128**

Sintassi:
COPY ["*file vecchio*"] [,D*n*] TO ["*file nuovo*"] [,D*n*] [,U*n*]

Abbreviazione:
coP

- Su un drive singolo, copia un file assegnandogli un nome nuovo.
- Su un drive doppio copia un file da un dischetto all'altro, anche con lo stesso nome.
- Sempre su un drive doppio, copia *tutti* i file da un dischetto a un altro.

Non è invece possibile eseguire copie tra unità (U) diverse. Si può usare invece BACKUP sul drive doppio.

Esempi:

```
COPY "soldi" TO "quattrini"
COPY "soldi",D0 TO "soldi",D1
COPY D0 TO D1
```

Il primo riscrive "soldi" sullo stesso disco con un nuovo nome: il secondo (in un drive doppio) lo copia sull'altro dischetto: il terzo ricopia tutto il dischetto sull'altro, sempre su un drive doppio. Un drive doppio accet-

ta due dischi all'interno della stessa unità (U8 è il default). Questi drive non sono in vendita in Italia, perciò è utile solo il primo esempio. Il dischetto C-128 DOS SHELL fornito insieme al calcolatore comprende un programma utilissimo per copiare più file da un disco a un altro. Il programma DIRTUT sul disco consente di duplicare file sullo stesso disco.

COS **Funzione** **20, 16, 64, 128**

Sintassi:
v=COS(*x*)

Abbreviazione:
nessuna

Restituisce il coseno dell'angolo *x* (espresso in radianti).

DATA **Istruzione** **20, 16, 64, 128**

Sintassi:
DATA *costante, costante, costante, ...,*

Abbreviazione:
dA

Mette a disposizione del programma una serie di dati. Questi sono contenuti in righe di programma ciascuna delle quali comincia con l'istruzione DATA. Queste righe possono essere in qualsiasi punto del programma. Non è necessario che siano nella parte del programma che viene eseguita: possono cioè essere anche dopo lo STOP finale.

```
List con virgolette
10 for t = 1 to 3
20 read a, a$
30 print a, a$
40 next
50 stop
100 data 1, "Maria", 2, "Teresa", 3, "Giovanna"
```

Sono state usate le virgolette perché se si scrive la riga

100 data 1, Maria, 2, Teresa, 3, Giovanna

quando si lista il programma, diventa:

100 data 1, rclraria, 2, instreresa, 3 chr$iovanna

Ciò è dovuto alla presenza delle maiuscole, e accade anche nelle REM. La System Guide ignora questo fatto che rende difficile la lettura del listato, ma, con o senza virgolette, il RUN produce:

```
1      Maria
2      Teresa
3      Giovanna
break  in line 50
```

Nella riga 100 è necessario che il primo, terzo, quinto... dato siano numerici (perché A è una variabile numerica). Se al posto di Maria, Teresa, Giovanna, ci fossero numeri, il calcolatore li tratterebbe come stringhe. Una costante stringa non deve necessariamente avere le virgolette, se non contiene i caratteri spazio, due punti, virgola. Se non c'è nessuna costante tra due virgole il calcolatore legge uno zero oppure un carattere nullo (""), rispettivamente per una variabile numerica o per una stringa. Se cerchiamo di leggere più dati di quanti non ce ne siano (nell'esempio, se t fosse >3), riceviamo un OUT OF DATA ERROR. Si vedano anche READ e RESTORE.

DCLEAR — Istruzione — 128

Sintassi:
DCLEAR [D*n*] [,U*n*]

Abbreviazione:
dclE

Simile all'istruzione INITIALIZE del DOS: chiude tutti i canali sul drive indicato (unità U8 se non è indicato diversamente). NON chiude i file aperti: chiude solo i canali; è quindi molto consigliabile chiudere prima i file (si veda DCLOSE).

DCLOSE — Istruzione — 128

Sintassi:
DCLOSE [# *file*] [,U*n*]

Abbreviazione:
dC

Chiude il *#file* specificato su drive e unità specificati: se *#file* non è specificato, chiude tutti i file su drive e unità specificati. Senza nessun argomento, chiude tutti i file sull'unità 8.

DEC — Funzione — 128

Sintassi:
n=DEC(*hex$*)

Abbreviazione:
nessuna

Dà il valore in notazione decimale di un valore esadecimale espresso come stringa:

```
A$=HEX$(15): ?A$
F
PRINT DEC (a$)
15
PRINT DEC("A")
10
```

Il valore 15, espresso in codice esadecimale è infatti F (10=$A, 11=$B, 12=$C, 13=$D, 14=$E, 15=$F e 16=$10). Il segno $ indica che il numero che segue è in forma esadecimale: questo tuttavia non è usato per la funzione DEC. Si veda anche la voce HEX$. Il Menu Comandi Vari comprende il programma hex.dec, che converte da HEX a DEC e viceversa.

DEF FN — Istruzione — 20, 16, 64, 128

Sintassi:
DEF FN *nome*(*variabile*)=*espressione*

Abbreviazione:
nessuna

Evita di far riscrivere, in un programma, complessi calcoli che devono essere ripetuti molte volte. Per esempio:

```
10 def fn a(x)=10*x
20 print fn a(9)
30 zz=1
40 print fn a(zz)
```

produce

```
90
10
ready
```

Il nome è A; potrebbe essere qualsiasi variabile numerica di una o due lettere (AB, XY, per esempio), ma NON una variabile intera (a%, per esempio). Una volta definita FN A(x) come nella riga 10, qualsiasi valore o variabile numerica sarà trattata come x nella parte della riga 10 che segue il segno =, anche se x compare più volte. L'uso di x nella riga 10 è "locale"; x può essere usato liberamente e indipendentemente nel resto del programma. DEF FN non può essere usato con funzioni stringa.

DELETE **Comando** **16, 128**

Sintassi:

DELETE *n*
DELETE −*n*
DELETE *n1*−*n2*
DELETE *n*−

Abbreviazione:

deL

Solo in modo diretto. Le quattro forme cancellano, rispettivamente:

- l'unica riga *n* specificata
- tutte le righe fino a riga *n*
- tutte le righe tra riga *n1* e riga *n2*
- tutte le righe a partire da riga *n*.

Le righe specificate sono comprese tra quelle da cancellare.

DIM **Istruzione** **20, 16, 64, 128**

Sintassi:
DIM *variabile* (*n*)
DIM *variabile* (*n, variabile, variabile, variabile,...*)

Abbreviazione:
dI

Determina il numero di elementi in una matrice di variabili tutte aventi lo stesso nome generale.
Supponiamo di avere una tabella del tipo seguente:

NOME	ETA'	SESSO	MANSIONI
Maria	22	f	segretaria
Giovanni	89	m	fattorino
Pierino	7	m	direttore

In questo caso è divisa in 4 colonne e 3 righe (per fare un esempio breve, ma potrebbe anche essere di molte più colonne e/o righe; le dimensioni sono limitate solo dalla memoria del calcolatore). Possiamo raggruppare tutti questi dati in una sola variabile, A$, scrivendo:

10 dim a$(4,3)

In questa matrice poi inseriremo (nei vari modi possibili) i vari dati (con INPUT, DATA...): qui si usa il metodo più leggibile per questa illustrazione, ma in pratica è meglio usare un'istruzione READ DATA:

```
dim.list
10 dim a$(4, 3)
20 a$(1,1) = "Maria    "
21 a$(1,2) = "Giovanni"
22 a$(1,3) = "Pierino "
23 a$(2,1) = "22"
24 a$(2,2) = "89"
25 a$(2,3) = " 7"
26 a$(3,1) = "f"
27 a$(3,2) = "m"
28 a$(3,3) = "m"
29 a$(4,1) = "segretaria"
30 a$(4,2) = "fattorino"
31 a$(4,3) = "direttore"
50 print "s"
```

E poi una routine per stampare ordinatamente la tabella:

```
110 for x=1 to 3
110 : for w=1 to 4
120 : print a$(w,x); " ";
130 :next w
140 print
150 next x
```

La riga 120 in questo esempio senza aiuto non incolonnerebbe molto bene i dati: sarà meglio usare PRINT USING o CHAR per formattare la stampa (l'alternativa è fare in modo che tutte le stringhe siano della stessa lunghezza, come in questo esempio). Esiste anche a$(0,0); per intestare la tabella:

```
a$(0,0) = "NOME"
a$(0,1) = "ETA' "
a$(0,2) = "SESSO"
a$(0,3) = "MANSIONI"
```

Inoltre non è necessario usare DIM se il numero di elementi è inferiore a 11. Ricordando che il numero degli elementi parte da 0, questo significa che il primo DIM necessario è DIM A$(11); questo crea infatti un array di 12 elementi. Se non lo specifichiamo, il calcolatore dimensiona a$ con 11 elementi (0-10).

Esempi:

DIM X$(3) crea una matrice-stringa (o array) di 4 elementi
DIM Z$(19,19) crea una matrice-stringa bidimensionale di 400 elementi (20×20)
DIM P$ (19,19,1) è tridimensionale con 20×20×2=800 elementi.
DIM a (9,9,9) matrice numerica di 1000 elementi.

Quando è possibile usare una variabile numerica intera (a%, per esempio), la memoria richiesta per la matrice si riduce al 40% di quella necessaria per una funzione numerica normale. Anche la velocità di esecuzione aumenta con le variabili intere.
Attenzione: l'istruzione DIM per ciascuna variabile deve essere letta dal calcolatore una sola volta nel corso del programma. NON si può ridimensionare una matrice, se non eseguendo in precedenza un CLR. È pertanto molto meglio raggruppare tutte le DIM insieme all'inizio del programma. In alternativa metterle in una subroutine alla fine del programma e poi usare un GOSUB all'inizio per andarle a leggere una sola volta.

In certe versioni (non Commodore) del BASIC, non è possibile creare matrici di stringhe e DIM serve per definire il contenuto di una stringa. In tutte le versioni CBM del BASIC, questa possibilità non esiste.

DIRECTORY **Comando** **16, 128**

Sintassi:
DIRECTORY [D*n*] [,U*n*] [,*stringa di ricerca*]

Abbreviazione:
diR

Elenca il direttorio del dischetto sul drive e sull'unità specificati (default 0, 8).
Caratteri speciali nella *stringa di ricerca*:

? un carattere qualsiasi
* qualsiasi numero di caratteri (deve essere alla fine)
= seleziona il *tipo* di file (v. esempi)

DIRECTORY "A* elenca tutti i file che cominciano con A
DIRECTORY "????.BAS elenca tutti i file che cominciano con 4 caratteri seguiti da un punto e dalle lettere BAS. Elencherà quindi FILE.BAS, PROG.BAS, e così via.
DIRECTORY "* =P darà invece tutti i file-programma
DIRECTORY "* =S darà tutti i file sequenziali
DIRECTORY "A* =P darà invece tutti i programmi che cominciano con A. E così via.

Il comando è identico a CATALOG. Si può usare in programma, come per esempio in SALVAGRAFICI e vari altri programmi del disco. Si veda anche il Capitolo 3 e il tutorial BASIC DISCO.

DLOAD **Comando** **16, 128**

Sintassi:
DLOAD "*nomefile*"[,D*n*] [,U*n*]

Abbreviazione:
dL

Carica un programma da disco. In modo diretto carica il programma sen-

za metterlo in esecuzione. In un programma, carica e passa l'esecuzione del programma a quello nuovo senza azzerare le variabili: RUN, invece, elimina le variabili e mette in esecuzione il nuovo programma.

DLOAD "programma" carica "programma"
DLOAD "* carica il primo programma trovato sul disco.

DO... LOOP... WHILE/UNTIL — Istruzione — 16, 128

Sintassi:
DO [WHILE/UNTIL *condizione*]...[EXIT]...LOOP [WHILE/UNTIL]

Abbreviazione:
DO=nessuna, UNTIL=uN, WHILE=wI

Analoga a FOR... NEXT. WHILE="mentre": UNTIL="fino a quando". EXIT="uscita"

```
10 do until x=20
15 :   print x
20 :   x=x+1
30 loop
```

stampa x fino a quando x=20; x non verrà stampata nemmeno una volta se è già maggiore di 20.

```
10 do
15 :   print x
20 :   x=x+1
30 loop until x=>20
```

In questo esempio, invece, x verrà stampata almeno una volta, anche se x>20. E in entrambi gli esempi avremo lo stesso risultato se sostituiamo "UNTIL X=20" con "WHILE X≤21".

```
10 do
15 :   print x
20 :   x=x+1
25 :   get a$: ifa$="*" then exit
30 loop
```

In questo esempio l'operazione verrà ripetuta ad infinitum se non verrà premuto il tasto *. Naturalmente EXIT (uscita) può essere usata anche

insieme a UNTIL o WHILE.
Il Capitolo 4 di questo volume e il programma omonimo forniscono una discussione più ampia, e numerosi altri programmi usano questa nuova e importante istruzione.

DOPEN **Istruzione** **128**

Sintassi:
DOPEN# *n*,"*nomefile* [,S|P]" [,L*n*] [,D*n*] [,U*n*] [,W]

Abbreviazione:
dO

Apre un file su disco per lettura e/o scrittura. "S" apre un file sequenziale, mentre per aprire un file relativo non si specifica né S né P, ma L*n* dove *n* è la lunghezza del record. ",S" e ",P" devono stare dentro le virgolette del nome del file.
L'essenziale è nei seguenti esempi:

DOPEN#3, "appendice" (apre "appendice" in lettura)
DOPEN#3, "appendice",w (apre "appendice" in scrittura (write))
DOPEN#1, "database", L35 (apre il file relativo "database" con record lungo 35 byte).

Il secondo esempio apre un file sequenziale: non è necessario specificare S né per la lettura né per la scrittura. Se si omette il W, il calcolatore sarà pronto per un'operazione di lettura.
Un file relativo è aperto sia per la lettura che per la scrittura. Rispetto al DOS del C-64 bisogna tener presente che la W va inserita dopo una virgola e senza virgolette. Chi era abituato al vecchio sistema, anche se quello nuovo è più comodo, rischia di sbagliare. Non accade niente di grave, ma ovviamente il file non viene scritto. Infine, tener presente che, se il nome di un file è contenuto in una variabile (A$, per esempio), la sintassi è come segue:

A$="APPENDICE": DOPEN#3 (A$)+",W"

DRAW **Istruzione** **16, 128**

Sintassi:
DRAW [*fonte colore*],*x1*,*y1* [TO *y2*,*y2*]...
oppure DRAW TO *xn*, *yn*

Abbreviazione:
dR

fonte colore è come per CIRCLE o BOX. Le coordinate *x1, y1* e *x2, y2* sono rispettivamente punto di partenza (influenzato da SCALE) e punto d'arrivo. Senza quest'ultimo, si disegna un punto (un solo pixel).

DRAW 1,100,100 un punto
DRAW 1,50,100 TO 100,100 una linea
DRAW 1,100,100 TO 100,150 TO 150,150 TO 150,100 TO 100,100
un rettangolo.

La System Guide omette di dare un'informazione tutt'altro che trascurabile: non è necessario indicare alcunché prima di TO. In altre parole, se dopo l'ultimo esempio inseriamo:

DRAW TO 0,0

si disegnerà una riga che, partendo dall'angolo superiore a sinistra del rettangolo (perché è a quel punto che si trova il pixel cursor dopo averlo disegnato), termina all'angolo superiore sinistro dello schermo. Quindi la parte prima di TO è da usare all'inizio di una serie di DRAW (per evitare che la prima riga parta da 0,0) oppure quando vogliamo iniziare un nuovo disegno senza una riga che colleghi il nuovo con il vecchio. Un capitolo di questo libro tratta più ampiamente l'argomento: il disco ha un tutorial (Lato 1) e tre programmi (Lato 2; descritti nel capitolo introduttivo): DRAW, JOYDRAW e MULTIDRAW.
Un altro fatto non documentato è la possibilità di usare *coordinate relative* come per MOVSPR, anche se non sono ammessi valori negativi.

DRAW TO+22,+66: disegna una riga da *x,y* a *x*+22, *y*+66
DRAW TO+45; 90: una riga da *x* a *x*+45 a un angolo di 90 gradi

Questo vale anche per altri comandi grafici.

DS **Funzione** **16, 128**

Sintassi:
PRINT DS

Abbreviazione:
nessuna

Dà il numero corrispondente a un errore verificatosi in relazione a operazioni su disco (si veda l'Appendice B della System Guide). È naturalmente possibile usarlo anche in programma, per esempio in corrispondenza con TRAP. PRINT DS e PRINT DS$ fermano il lampeggio della spia d'errore del drive e stampano il numero e la stringa corrispondenti all'errore.

DS$ **Funzione** **16, 128**

Sintassi:
PRINT DS$

Abbreviazione:
nessuna

Fornisce il messaggio a parole corrispondente all'errore che ha il numero DS. Come DS, si può usare in programma, ma la sua utilità più immediata è in modo diretto. L'Appendice B della System Guide dà una spiegazione di ciascun messaggio.

DSAVE **Comando** **16, 128**

Sintassi:
DSAVE"*nomefile*" [,D*n*] [,U*n*]

Abbreviazione:
dS

Salva un programma su disco. Utilissimo. Si veda anche DLOAD. Se si salva un programma il cui nome è contenuto in una variabile, questa va indicata tra parentesi.
Si vedano anche le voci SAVE e BSAVE. Per sostituire un file esistente, usare SCRATCH e (eventualmente) COLLECT prima di DSAVE. Evitare la forma DSAVE "*@nomefile*".

DVERIFY **Comando** **128**

Sintassi:
DVERIFY "*nomefile*" [,D*n*] [,U*n*]

Abbreviazione:
dV

Verifica un programma su disco. È molto veloce con il 1571. Un VERIFY ERROR si presenterà se, dopo aver salvato il programma, si alloca o si disalloca un'area grafica. Ciò non significa necessariamente che il programma sia errato: sarà sufficiente eseguire GRAPHIC CLR se il programma era stato salvato quando non esisteva l'area grafica, o GRAPHIC 1: GRAPHIC 0 se invece era stato salvato mentre occupava l'area riservata al BASIC quando è attivo lo schermo grafico. DVERIFY non è utilizzabile per dati salvati con BSAVE (per sprite, per esempio, o per il bit map dello schermo grafico — salvato come nel programma SALVAGRAFICI). Per questa operazione il comando è:

VERIFY "*nomefile*",8,1

EL — Variabile del sistema — 16, 128

Sintassi:
n=EL

Abbreviazione:
nessuna

La variabile contiene il numero della riga in cui si è verificato l'ultimo errore. In modo diretto può essere utile se per qualche motivo non si è riusciti a vedere un messaggio come "SYNTAX ERROR IN..."; in programma (particolarmente nelle routine raggiunte da TRAP) è generalmente usata con una espressione IF...THEN

```
1000 if el=210 then...
```

ELSE vedi IF.

END — Istruzione — 20, 16, 64, 128

Sintassi:
END

Abbreviazione:
nessuna

Indica al calcolatore la fine del programma: differisce da STOP nel fatto che non dà messaggi: ricompare il cursore dopo la parola READY. Con

STOP si ha invece "BREAK IN...". Si può usare CONT per riprendere l'esecuzione.

ENVELOPE **Comando** **128**

Sintassi:
ENVELOPE *n* [,*atk*] [,*dec*] [,*sus*] [,*rel*] [,*wf*] [,*pw*]

Abbreviazione:
eN

Trattato anche nel Capitolo 13. Envelope significa "inviluppo"; definisce le caratteristiche fondamentali che danno il timbro al suono di uno strumento musicale.

n = numero dell'inviluppo (0-9)
atk = attacco: velocità (0-15) con cui il suono raggiunge il massimo volume
dec = declino (0-15): dopo l'attacco il volume, specie di una corda pizzicata, declina a un livello "medio" (definito a sua volta da *sus*)
sus = sostentamento (0-15): il volume del suono per la maggior parte del tempo in cui si sente; il livello "medio".
rel = rilascio (0-15): la velocità con cui declina dal livello *sus* a zero. Estinzione.
wf = forma d'onda (waveform). Numero da 0 a 4. Le forme d'onda sono illustrate molto bene alla pagina 7-12 della System Guide, tranne "ring modulation".
pw = larghezza d'impulso (pulse width): distanza nel tempo tra due impulsi con la forma d'onda 2. Valori da 0 a 4095.

Misericordiosamente ci sono già 10 envelope definiti per 10 diversi strumenti, anche se non tutti forse corrispondono molto bene al nome. Entrare in una discussione molto dettagliata su come usare i comandi ENVELOPE e FILTER richiederebbe un libro a parte.
Alcuni aspetti di questo comando sono discussi sommariamente nel capitolo dedicato a PLAY, ma la documentazione più ampia è contenuta nel programma ENVELOPE sul dischetto (Menu Musicale). Questo presenta tutti i suoni ottenuti variando a uno a uno i singoli parametri di un ENVELOPE, anche se non è ovviamente possibile illustrare tutte le possibili combinazioni di tutti i valori di tutti i parametri (totale $15 \times 15 \times 15 \times 15 \times 4 \times 4096$). Fornisce una vastissima gamma di possibilità musicali. Per un esempio si veda anche il programma BACH2 sullo stesso menu.

ER — Variabile del sistema — 128

Sintassi:
v=ER

Abbreviazione:
nessuna

Restituisce il numero corrispondente all'errore che si è verificato. Nelle routine attivate da TRAP sono utili le espressioni del tipo

```
2000 if er=13 then restore: resume
```

Si vedano la voce ERR$ e RESUME.

ERR$ — Funzione — 128

Sintassi:
a$=ERR$(*n*)

Abbreviazione:
eR

Stampa il messaggio d'errore corrispondente a *n* (Appendice A, System Guide).
Se si è verificato un errore, il numero dell'errore è contenuto nella variabile ER; quindi

```
PRINT ERR$(ER)
```

dà il messaggio relativo all'errore attuale. In altre parole, i messaggi d'errore sono tutti contenuti nella matrice ERR$ (di 41 elementi). Perciò

```
10 for t=1 to 41: print err$(t): next
```

stamperà l'elenco completo. Non esiste ERR$(0).

EXIT — Istruzione — 16, 128

Sintassi:
EXIT

Abbreviazione:
exI

Fa uscire da un loop iniziato con DO. Vedi anche DO...LOOP.

EXP **Funzione** **20, 16, 64, 128**

Sintassi:
x=EXP(*n*)

Abbreviazione:
eX

Dà il valore di e elevato alla *n*-esima potenza.

FAST **Comando** **128**

Sintassi:
FAST

Abbreviazione:
nessuna

Particolarmente utile per chi ha il monitor a 80 colonne; con le 40 colonne invece, si perde l'immagine sul video mentre il calcolatore è in modalità FAST. Normalmente il C-128 opera a una velocità interna di 1 MHz, che è decisamente poco rispetto ai 6-8 MHz di calcolatori come l'IBM PC AT o lo stesso Commodore Amiga.
FAST porta la velocità a 2 MHz, raddoppiando in pratica la velocità di esecuzione delle operazioni interne (non influenza cioè la velocità di trasmissione/ricezione di dati verso le periferiche). Poiché anche la grafica appartiene al mondo delle 40 colonne, essa è invisibile durante il FAST. Ciò nonostante, la grafica funziona: si può accelerare la creazione di un grafico complesso, lasciando lo schermo vuoto finché il programma non raggiunga un comando SLOW. Per esempio RENUMBER può essere una delle operazioni più lente che esistono, se il programma è lungo. Si può accelerarlo con FAST, anche se non si dispone del monitor a 80 colonne; l'operazione avviene in modo diretto:

```
FAST: RENUMBER
READY
SLOW
```

Il comando SLOW deve essere dato "al buio" dopo l'operazione.
All'inizio di un programma:

0 if rgr(0)=4 then FAST

inserisce automaticamente la velocità di 2 MHz se è attivo lo schermo a 80 colonne. I valori di RGR(0) superiori a 4 indicano che è attivo lo schermo a 80 colonne.

FETCH **Istruzione** **128**

Sintassi:
FETCH *#byte, intsa, expsa, expb*

Abbreviazione:
fE

Legge *#byte* di dati da un'espansione di memoria. Dopo aver indicato il numero dei byte da leggere, si indicano:

intsa = indirizzo di partenza nella memoria "interna" (cioè non di espansione); i dati verranno trasferiti qui dall'espansione.
expsa = indirizzo nell'espansione in cui deve cominciare la lettura;
expb = numero del banco di memoria in cui si trovano i 64 K di espansione (ciascun banco comprende 64 K di memoria, indirizzi 0–65535). Si veda anche il capitolo sulle espansioni di memoria.

FILTER **Istruzione** **128**

Sintassi:
FILTER [*t*] [,*b*] [,*m*] [,*a*] [,*ris*]

Abbreviazione:
fI

Modifica un suono definito con un ENVELOPE: privilegiando certe frequenze ed attenuando altre ne cambia il timbro o risonanza. Determina il parametro "X" di una stringa per PLAY;

t = frequenza di taglio (frequenza di riferimento per *b, m, a*)
b = passa-basso 1=on, 0=off: attenua le frequenze più alte di *t*.

m = passa banda 1=on, 0=off: lascia passare una banda centrale di frequenze, attenuando quelle più alte e più basse.
a = attenua le frequenze inferiori a *t*. 1=on, 2=off.
ris = risonanza: cambia la nitidezza del suono (0-15).

L'istruzione FILTER richiede una certa pratica per ottenere l'effetto desiderato. L'approccio sperimentale è il più efficace. Si veda anche PLAY.

FNxx — Funzione — 20, 16, 64, 128

Sintassi:
v=FN*xx*(*X*)

Abbreviazione:
nessuna

Restituisce il valore risultante da una funzione definita dall'utente: si veda DEF FN. Le lettere *xx* si riferiscono appunto alla funzione definita con DEF FN. *X* è un argomento (valore numerico). Si veda anche DEF FN.

FOR... TO... STEP:... NEXT — Istruzione — 20, 16, 64, 128

Sintassi:
FOR *x*=*n1* TO *n2* [STEP *i*] *.../serie di istruzioni/...*: NEXT

Abbreviazione:
FOR=fO, NEXT=nE, STEP=stE

Da molti punti di vista è l'istruzione più importante del BASIC, e infatti esiste in tutte le versioni.
Partendo da FOR, ogni volta che il loop arriva a NEXT, incrementa o decrementa *n1* del valore *i* (se STEP è omesso, *i*=1). Quando *n1*=*n2* il programma esce dal loop e passa al comando immediatamente successivo. La forma base è:

```
10 for t=1 to 10
20 :    print t
30 next
```

In 30 NEXT="NEXT T"... poiché il loop è singolo si può omettere T.

```
10 for t=10 to 0 step -2
20 :    print t
30 next
```

La riga 10 del primo esempio contiene (non scritto) STEP+1, che è il default. Nel secondo, STEP −2 decrementa T di due a ogni passaggio.
Loop annidati:

```
10 for t=1 to 10
15 :  for u=10 to -10 step -2
20 :    print "t=";t
30 :    print "u=";u
40 :  next u
50 next t
```

Non c'è limite al numero di loop che si possono inserire l'uno dentro l'altro. NEXT (l'istruzione di chiusura) funziona in ordine opposto. Cioè il primo loop ad aprirsi deve essere l'ultimo a chiudersi, e così via. Al posto delle righe 40 e 50 si potrebbe scrivere soltanto:

```
40 next u, t
```

Si possono usare variabili numeriche nei loop:

```
FOR T=X TO Y STEP Q.....
```

ma non variabili stringhe o intere (come T$ o Y%).

FRE   Funzione   20, 16, 64, 128

Sintassi:
  x=FRE(*n*)

Abbreviazione:
  nessuna

Restituisce il numero di byte liberi (FREE) nel banco *n*.

```
PRINT FRE(0)
```

dà il numero di byte liberi nel banco 0 (riservato ai programmi in BASIC).

```
PRINT FRE(1)
```

dà i byte liberi per le variabili del programma.
Se le variabili vengono cambiate molte volte nel corso di un programma, è utile sapere che i vecchi valori rimangono provvisoriamente e viene allocata una nuova area di memoria alla variabile. Il sistema operativo "raccoglie" periodicamente lo spazio inutilmente occupato. Poiché questa operazione può richiedere diversi secondi, se si teme che possa provocare una pausa imbarazzante, è possibile provocare un "garbage collect" in un punto meno delicato del programma, con

```
X=FRE(1)
```

dove X è una variabile qualsiasi. L'operazione serve solo per il banco 1, che contiene le variabili.
Sul C-64 l'argomento *n* era un "dummy", cioè qualsiasi valore di *n* dava lo stesso risultato.

```
0 if fre(1)=fre(0) then go 64
```

è un modo per far sì che il C-128 passi automaticamente al modo C-64.

GET — Istruzione — 20, 16, 64, 128

Sintassi:
GET *variabile*

Abbreviazione:
gE

Guarda nel buffer della tastiera per vedere se un tasto è stato premuto: se sì, la prima variabile della lista assume il valore del tasto. A differenza di INPUT, GET non si ferma se non c'è niente.
Sul VIC-20 e sul C-64, GET si usava come "trappola" per creare una pausa nel programma:

```
10 get a$: if a$="" then 10
```

Sul C-16/Plus/4 e sul 128 GETKEY A$ fa la stessa cosa. GET continua a essere utile invece per ricevere istruzioni facoltative dall'utente.

```
10 do
20 :    play "stringa musicale"
```

```
30:    get a$: if a$="!" then exit
40:    if a$="*" then play "stringa musicale 2": a$=" "
50 loop
```

Qui "!" termina il loop chiuso, mentre "*" cambia la musica. Se invece l'utente non fa niente, o preme altri tasti, il programma continua all'infinito. Si può premere un tasto in qualsiasi momento.
GET attende sia una stringa di un carattere, sia un valore numerico da 0 a 9. Se attende a$, si può premere qualsiasi tasto. Ma se attende una variabile numerica e l'utente preme un altro tasto, il programma si fermerà con TYPE MISMATCH ERROR. Usare l'istruzione TRAP per evitare disastri. L'esempio serve a illustrare la differenza tra GET e INPUT: mentre INPUT ferma il programma fino a quando l'utente non abbia dato le informazioni richieste, con GET (generalmente all'interno di un loop), continua a fare altri lavori fino a quando non viene premuto un tasto.

GET# — Istruzione — 20, 16, 64, 128

Sintassi:
GET# *nf, variabile*

Abbreviazione:
gE

Legge dal file *#nf* un carattere per volta (variabile stringa o numerica). Identico a GET, solo che GET legge la tastiera. È chiaro che GET# deve essere preceduto da un OPEN. Non è ammesso uno spazio tra GET e #. GET# è molto utile per leggere file complessi, in cui INPUT# potrebbe produrre delle stringhe troppo difficili da trattare.
Il programma TYPE sul Lato 1 del disco ne è un esempio. Si basa su una struttura del tipo:

```
10 do until st
20:    get#1,a$: print a$
30 loop
```

TYPE, però (come anche FIND), è più complesso, essendo strutturato in modo da non stampare certi caratteri non corrispondenti a "testo normale", che avrebbero effetti negativi se stampati sullo schermo (si veda il programma): in questo modo è possibile ricercare stringhe anche all'interno di un file di programma. La funzione ST assume il valore 64 quando si raggiunge la fine del file: così la riga 10 evita che il programma cerchi di leggere oltre la fine del file stesso. ALFASORT usa una routine

analoga per contare il numero di stringhe in un file sequenziale: conta il numero delle ricorrenze di CHR$(13) [RETURN]: questo è un sistema primitivo che non sempre dà il risultato richiesto ed è meglio conoscere il numero prima; ma in un programma non in linguaggio macchina era l'unica soluzione non eccessivamente lenta.

GETKEY **Istruzione** **16, 128**

Sintassi:
GETKEY *variabile*

Abbreviazione:
getkE

A differenza da GET, GETKEY attende finché non venga premuto un tasto. È quindi un ottimo modo di fare sospendere il programma finché l'utente non sia pronto a continuare.
A differenza da INPUT, GETKEY non presenta un ? e il cursore non è presente.

```
100 print "Premere un tasto per continuare"
20 getkey a$
30 print "È stato premuto un tasto"
```

Accetta variabili stringa o numeriche, con le stesse modalità di GET. Può attendere più di una variabile:

```
10 getkey A$, A, B$, b...
```

GO64 **Istruzione** **128**

Sintassi:
GO64

Abbreviazione:
nessuna

Sia da programma, sia in modo diretto, questa istruzione fa sì che il 128 diventi, da tutti i punti di vista pratici, un C-64. Il calcolatore riparte da "freddo" (cioè come all'accensione) in modo C-64. Pertanto ogni programma, variabile, ecc., in memoria, viene perduto. Per questo il C-128 presenta il messaggio ARE YOU SURE? (Sei sicuro?), come con SCRATCH. Pre-

mere Y (yes) solo se si è sicuri. Qualsiasi altro tasto lo annulla. Da programma, esegue l'ordine subito e senza presentare la domanda. Come NEW, è da usare normalmente da tastiera. Se si vuole usare un disco BOOT per un programma in BASIC 2.0, il sistema può passare automaticamente in modo 64 se il programma boot è nella forma

```
10 go64
```

Un altro modo di passare al modo 64 è di tenere premuto il tasto COMMODORE mentre si accende il sistema. Esiste una differenza che può essere scomoda oppure utile a seconda delle esigenze: in questo caso il drive 1571 si comporta come il 1541 non solo per quanto riguarda la velocità (minore); tratta il disco a doppia faccia come se fosse a faccia singola e il direttorio riporta l'elenco dei file esistenti sul primo lato soltanto. Con GO64, invece, il 1571 legge entrambi i lati. Il ritorno al modo 128 è possibile soltanto premendo RESET o spegnendo il sistema.

GOSUB **Istruzione** **20, 16, 64, 128**

Sintassi:
GOSUB *riga*

Abbreviazione:
goS

Va alla *riga* specificata, esegue un sottoprogramma (subroutine), e quando incontra l'istruzione RETURN, ritorna all'istruzione successiva al GOSUB

```
10 print "Inizio programma"
20 gosub1000
30 .. [resto del programma] ..
100 end
1000 print "Premere un tasto per continuare"
1010 getkey a$
1020 return
```

La riga 20 manda il programma alla riga 1000, che informa "Premere un tasto...". Se un tasto viene premuto alla riga 1010, ritorna alla 30.
Attenzione: una subroutine non deve essere messa in una posizione tale che il programma possa finirci dentro per errore. Perciò la riga 100 contiene END, per evitare che il programma vada alla 1000 senza aver trovato un GOSUB. In tal caso, infatti, se l'utente premesse il tasto, tutto si

fermerebbe con un RETURN WITHOUT GOSUB ERROR (RETURN senza GOSUB). È senz'altro meglio mettere tutte le subroutine dopo END, ma in alternativa (se immaginiamo che ci sia un altro blocco di programma a partire da 2000), si potrebbe scrivere:

```
100 gosub 1000: goto 2000
```

GOTO **Istruzione** **20, 16, 64, 128**

Sintassi:
GOTO *riga*

Abbreviazione:
gO

Molto semplice, ma non per questo meno importante. Un uso tipico è contenuto nell'ultimo esempio per GOSUB (sopra). Si noti, nel seguente esempio, che GOTO può essere sottinteso:

```
10 input "Un numero";a
20 if a=0 then 1000
30 if sgn(a)=-1 then 2000: else 3000
```

Qui THEN="THEN GOTO" ed ELSE=ELSE GOTO.
Un uso comodo di GOTO è quando vogliamo dare il RUN a un programma, senza cancellare le variabili dalla memoria. GOTO battuto in modo diretto e seguito da un numero di riga equivale a RUN seguito dallo stesso numero, solo che le variabili rimangono. Questo è prezioso mentre si sta collaudando un programma.

GRAPHIC **Istruzione** **16, 128**

Sintassi:
GRAPHIC *modo*, [,*clr*] [,*s*]
oppure GRAPHIC CLR

Abbreviazione:
gR

Seleziona il modo di presentare dati sullo schermo. *modo* può valere:

0 - testo a 40 colonne
1 - grafica normale ad alta risoluzione

2 - grafica con una finestra di testo normale
3 - grafica multicolor a bassa risoluzione
4 - multicolor con finestra
5 - testo a 80 colonne.

clr è un numero (0 o 1): 1 significa ripulire lo schermo; 0 significa lasciare il grafico esistente. Nei modi grafici 1 e 2, inoltre, il colore dello sfondo rimane inalterato, se non si indica 1 per questo parametro, anche se si esegue un'istruzione COLOR 0. Invece, nei modi grafici 3 e 4, COLOR 0, *n* cambia il colore dello sfondo in qualsiasi momento.

s nei modi GRAPHIC 2 e GRAPHIC 4, definisce una finestra in fondo allo schermo, che consente di vedere una parte dello schermo di testo a 40 colonne. Il numero *s* specifica da quale delle 25 righe (0-24) deve cominciare la finestra. GRAPHIC 2 o GRAPHIC 4, senza valore per *s*, apre la finestra alla 20ª riga (*s*=19). Questo è molto utile quando si disegna in modo diretto, o quando si sta collaudando un programma. Se si desidera una finestra diversa dal default e non si desidera pulire lo schermo, inserire solo la virgola di [,*clr*]:

```
GRAPHIC 2,,15
```

apre una finestra alla sedicesima riga, senza toccare il contenuto dello schermo. Si torna a vedere tutto il grafico eseguendo GRAPHIC 1 (o GRAPHIC 3).

L'istruzione GRAPHIC è ovviamente necessaria prima di visualizzare un qualsiasi lavoro grafico ottenuto con CIRCLE, BOX, DRAW, ecc. Ma se non si vuole che l'utente veda il grafico mentre viene disegnato, GRAPHIC può essere eseguito quando il disegno è già pronto, purché l'istruzione GRAPHIC sia stata usata almeno una volta, senza aver poi usato GRAPHIC CLR, cioè se è già stata riservata l'area di lavoro necessaria per la grafica. Il comando FAST aumenta notevolmente la velocità di esecuzione.

Nota: in un certo senso i modi grafici non sono 6 ma 8 (numerati da 0 a 7). Infatti, la funzione RGR (la System Guide lo ignora) dà valori da 0 a 5. I due valori ignorati sono:

RGR (0)=6 Sono attivi sia lo schermo GRAPHIC 1 che lo schermo GRAPHIC 5 (80 colonne). Con due monitor (o un monitor RGBI e un TV) si ha un'alternativa strana ma molto comoda rispetto a GRAPHIC 2.

RGR (0)=8 GRAPHIC 3 più GRAPHIC 5. Anche in questo caso si può lavorare con 2 monitor.

Per ottenere questi due valori di RGR, però, non si può usare GRAPHIC 6 o GRAPHIC 7. Bisogna selezionare prima uno dei due schermi di testo (0/5), poi uno schermo grafico.
L'istruzione GRAPHIC CLR (con o senza spazio) è diversa dal [,*clr*] già citato. Quest'ultimo elimina semplicemente il disegno precedente. È sempre necessario all'inizio di un programma, perché la prima volta che si entra in modo grafico lo schermo è sempre pieno di rifiuti.
GRAPHIC CLR, invece, fa una cosa più radicale. Quando si entra in uno dei modi grafici numerati da 1 a 4, il calcolatore assegna oltre 9 K di memoria al "bit map" per lo schermo grafico. Questa sezione di memoria parte dall'inizio dell'area normalmente riservata al BASIC. Il programma già in memoria viene pertanto traslocato in modo da liberare l'area. Se non serve più l'area grafica, e serve avere a disposizione molta memoria per il programma, GRAPHIC CLR annulla il trasloco, e l'inizio del programma torna a coincidere con l'inizio del BASIC.

Esempi:

GRAPHIC 0	torna allo schermo normale (40) (in modo diretto si può anche premere ESC-X)
GRAPHIC 1,1	passa a grafica normale, pulisce lo schermo
GRAPHIC 2,0,15	passa a grafica normale, non pulisce lo schermo, finestra a partire dalla 16ª riga
GRAPHIC 5	passa a 80 colonne (in modo diretto si può anche premere ESC-X).

Si vedano i programmi DRAW, JOYDRAW e MULTIDRAW in particolare, oltre ai vari programmi dedicati ai singoli comandi grafici.

GSHAPE **Comando** **16, 128**

Sintassi:
GSHAPE *v*$, [*x,y*] [,*modo*]

Abbreviazione:
gS

Posiziona sullo schermo grafico una forma grafica precedentemente salvata nella stringa *v*$, alle coordinate *x,y* (influenzate da SCALE: se omesse il default è la posizione del pixel cursor). La stringa *v*$ (che può avere qualsiasi altro nome) può essere una stringa di sprite creata con SPRSAV, oppure una stringa grafica anche più lunga creata con SSHAPE: si rimanda a queste voci e al programma SHAPES (Lato 1) per una discussione più estesa.

HEADER **Comando** **16, 128**

Sintassi:
HEADER "*nomedisco*" [I*nd*] [,D*n*] [,U*n*]

Abbreviazione:
heA

Formatta un dischetto su drive e unità specificati. Sostituisce OPEN 15, 8,15, "NO:*nomedisco,nd*", usato su VIC-20 e C-64, ed è un po' più semplice. *nd* è qualsiasi coppia di caratteri, non necessariamente un numero.

HEADER "DISCO BELLO", IX9

I due caratteri *nd* possono essere contenuti in una variabile.
Per riciclare un dischetto già formattato, eliminando tutti i file e cambiandone il nome:

HEADER "*nuovonome*"

(senza *nd*) è più veloce. Come per SCRATCH e GO64 il calcolatore chiede:

ARE YOU SURE?

e si deve rispondere Y (yes) oppure N (no).

HELP **Comando** **16, 128**

Sintassi:
HELP

Abbreviazione:
heL

Si usa in modo diretto. Scrivere HELP RETURN, oppure battere il tasto HELP (in alto al centro) dà lo stesso risultato. Dà un'indicazione approssimativa della posizione di un errore in un programma. Quando un programma si interrompe, HELP presenta la riga colpevole (in negativo sullo schermo a 40 colonne, sottolineato su quello a 80 colonne). Il programma KEY sul disco fornisce un modo di riprogrammare il tasto HELP (che, come RUN, è un tasto funzione).

HEX$ **Funzione** **128**

Sintassi:
n$=HEX$(*n*)

Restituisce una stringa contenente il valore *n* espresso in notazione esadecimale, con *n*> 0 e < 65535 (da $0000 a $FFFF in notazione esadecimale).
Ecco un programma che converte valori da HEX a DEC e viceversa. Se un valore è preceduto da $, produce l'equivalente decimale; altrimenti fornisce il valore esadecimale.

```
hex.dec.list
5 trap2000
10 color 0, 16: color 4, 1: color 5, 1: color 6, 16
20 print chr$(14);" Hex/Dec"
25 print chr$(142)
30 print " inserire il valore da convertire.
40 print " se il valore e´ esadecimale, deve
50 print " essere preceduto da $
55 print " (massimo $ffff = 65535)"
60 input x$
70 if instr(x$,"$") then x$ = right$(x$,(len(x$)-1)):
   print " l´equivalente decimale di $";x$" e´":
   print " ";dec(x$):goto1000
80 print " l´equivalente esadecimale di " ;val(x$);"e´ ":
   print hex$(val(x$))
1000 print chr$(14);"Ripetere/ Menu": getkeya$:
     if a$ = "r" then run:
     else if a$ = "m" then run "str.menu": else end
1010 end
2000 print chr$(7);"valore errato":sleep5:run
ready.
```

Si veda anche la voce DEC.

IF...THEN [:ELSE] **Istruzione** **20, 16, 64, 128 (20, 64 senza ELSE)**

Sintassi:
IF *condizione* THEN [BEGIN] *comando* [:ELSE *comando*]

Abbreviazione:
nessuna

SE *condizione* ALLORA *comando* [:ALTRIMENTI *comando*]. Un'altra

istruzione fondamentale del BASIC e (con variazioni) in ogni linguaggio di programmazione. L'aggiunta di ELSE è preziosa.
Prima bisognava scrivere

```
10 if x=0 then y=1
20 if x<>0 then y=0
```

(o, naturalmente, qualche espressione molto più lunga). Con il 128 basta

```
10 if x=0 then y=1: else y=0
```

IF esegue l'operazione logica di stabilire se l'espressione che lo segue sia vera o falsa. Il risultato 0=falso, 1= vero. Così:

IF 1 THEN PRINT "UNO" farà stampare "UNO" tutte le volte.
IF X THEN PRINT "OK" è come scrivere IF X<>0 THEN PRINT "OK":

stamperà "OK" solo se X ha un valore diverso da 0.

IF X<>1 THEN PRINT "NO". Stamperà "NO" se X non è 1.

Si combina con molti operatori:

IF X=1 AND Y=2 THEN PRINT ...

richiede che X=1, Y=2

IF X=1 OR Y*N=21 THEN GOSUB ....

richiede che l'una OPPURE l'altra delle due espressioni (o entrambe) sia vera.

THEN seguito da un numero è come THEN GOTO seguito dallo stesso numero.

ELSE viene eseguito dal calcolatore solo se l'espressione dopo IF è falsa. Seguito da un numero è come scrivere ELSE GOTO seguito dal numero. ELSE deve essere nella stessa riga di programma; non avrebbe comunque senso metterlo nella riga successiva.

THEN può essere seguito da BEGIN. Quest'ultimo comando, potentissimo, crea una serie di righe di programma (che termina con l'istruzione BEND), tutte dipendenti dall'IF iniziale. Con questo si guadagna non solo tempo, ma anche chiarezza nella

lettura del programma. ELSE in questo caso deve seguire BEND. Si veda la voce BEGIN e la discussione nel Capitolo 4. Gli esempi della System Guide, per un lettore superficiale, potrebbero far pensare che IF...THEN serva solo per espressioni come:
IF *espressione* THEN *numero di linea.*

Ovviamente THEN può essere seguito da qualsiasi comando, istruzione, routine, eccetera.

```
100 if x=5 then y=2: print y: goto 50: else y=1: q=0: gosub 400
```

INPUT — Istruzione — 20, 16, 64, 128

Sintassi:
INPUT [*stringa;*] *variabile...*

Abbreviazione:
nessuna

Attende dalla tastiera una stringa o valore numerico seguito da RETURN. Le variabili attese devono essere separate da virgole:

```
10 input "Nome ed età"; n$, e
```

Le stringhe possono avere fino a 160 caratteri. Non possono contenere virgole né due punti.
A differenza di GETKEY, INPUT aggiunge un ? e il cursore continua a lampeggiare. Se sono richieste più variabili, la seconda e le successive sono attese con ??.
Se si desidera che INPUT funzioni senza punto interrogativo, è possibile usare INPUT#, nella forma:

```
5 open 3,0
10 print"Nome";:input#3, n$
```

Questo, invece, funziona per una sola variabile: se si richiedono due variabili come nell'esempio precedente, presenta un punto interrogativo per la seconda variabile.
Se si risponde a INPUT premendo semplicemente RETURN, la variabile (o le variabili) conservano il valore che avevano in precedenza.
INPUT e INPUT# sono utilizzabili solo all'interno di un programma.

Con il C-64, quando è attivo un INPUT, non esiste alcun modo di fermare il programma se non premendo STOP-RESTORE. Per il C-128, invece, premere STOP-ENTER per fermare il programma senza perdere le variabili in memoria.

INPUT # — Istruzione — 20, 16, 64, 128

Sintassi:
INPUT # *nf, variabile...*

Abbreviazione:
iN

Attende dati da un file (disco, nastro, modem, ecc., e anche dallo schermo). Il file deve essere aperto (v. OPEN e DOPEN).

```
10 dopen#1,"file": input#1, a$, b
```

legge una stringa e un numero da disco.

```
20 open1,1,0,"file": input#1, a$, b
```

esegue la stessa operazione, ma da nastro.
Non è ammissibile uno spazio tra INPUT e #; tra # e *nf* è opzionale.

INSTR — Funzione — 16, 128

Sintassi:
n=INSTR(*stringa1, stringa2* [*,iniz*])

Abbreviazione:
inS

Cerca *stringa2* all'interno di *stringa1* [a partire dalla posizione indicata da *,iniz*]. Restituisce il numero della posizione del byte nella *stringa1* dove inizia *stringa2*. Trattato nel Capitolo 6 di questo libro.

```
10 if instr (a$,"ok",10) then ...
```

In entrambi questi esempi, l'azione specificata dopo THEN avrà luogo se i caratteri "ok" esistono nella stringa dopo il nono byte (carattere).

```
A$="LA VISPA TERESA"
PRINT INSTR(A$,"TERESA")
10
```

Una funzione preziosissima per molti lavori di text processing, ecc. Si veda il programma INSTR.

INT **Funzione** **20, 16, 64, 128**

Sintassi:
v=INT(n)

Abbreviazione:
nessuna

Restituisce il numero intero corrispondente a n senza la parte decimale. Se il valore è negativo, dà invece il numero intero inferiore

```
X=1.5: PRINT INT(X)
1
X=-1.5: PRINT INT(X)
-2
```

JOY **Funzione** **16, 128**

Sintassi:
x=JOY(n)

Abbreviazione:
jO

Restituisce la posizione del joystick. Il numero n (1 o 2) è il numero del joystick. x darà

```
      1
   8     2
7     0     3
   6     4
      5
```

Se anche il "grilletto" o bottone di sparo è premuto, il valore di x è incrementato di 128.

Il programma JOYDRAW del dischetto illustra un modo per utilizzare questi valori. Notare che si deve usare la porta 2: la porta 1, infatti, serve anche per altre periferiche e non può essere usata dentro un loop contenente per esempio dei GET senza ricorrere a degli artifici. È buona norma non inserire il joystick con il calcolatore già acceso.
Infine si consiglia di utilizzare un joystick di buona qualità: soprattutto per programmi come JOYDRAW, che deve disegnare pixel per pixel, un joystick da quattro soldi è praticamente inutilizzabile. Questa osservazione è ancora più valida per il programma MULTIDRAW che, a scopo sperimentale, era stato compilato in p-code (pseudo-codice). Il loop che interpreta il joystick (identico a quello di JOYDRAW) era tanto più veloce che, con un joystick di media qualità, non si riusciva a controllarlo in tempo utile.

KEY **Istruzione** **16, 128**

Sintassi:
KEY [*n, stringa*]

Abbreviazione:
kE

Elenca le stringhe attualmente inserite per ciascun tasto. Ecco l'elenco dei valori di questi tasti al momento dell'accensione, prodotto inserendo in modo diretto la parola KEY RETURN:

```
key 1,"graphic"
key 2,"dload"+chr$(34)
key 3,"directory"+chr$(13)
key 4,"scnclr"+chr$(13)
key 5,"dsave"+chr$(34)
key 6,"run+chr$(13)
key 7,"list+chr$(13)
key 8,"monitor"+chr$(13)
```

Questi default sono in realtà d'utilità limitata, ma si possono riprogrammare (anche dall'interno di un programma), e possono essere resi utilissimi. CHR$(13) e CHR$(34) sono rispettivamente RETURN e le virgolette. Così, premere F7 equivale a battere da tastiera LIST e RETURN, e F2 stampa sullo schermo:

```
DLOAD"
```

lasciando poi all'utente il compito di indicare il nome del programma e, evidentemente, di battere RETURN.
CHR$(34) serve perché non è possibile definire una stringa usando un numero *dispari* di virgolette. NON potremmo, in altre parole, programmare il tasto 7 con

```
KEY7, "DLOAD" "
```

Quest'istruzione è discussa anche nel Capitolo 1, mentre il programma KEY (Menu Comandi Vari) non solo fornisce illustrazioni ma consente di riprogrammare anche i tasti funzione RUN e HELP.

LEFT$ — Funzione — 20, 16, 64, 128

Sintassi:
x$=LEFT$(*stringa,n*)

Abbreviazione:
leF

Restituisce una stringa contenente gli *n* caratteri più a sinistra in *stringa*:

```
10 a$="Teresa è bella"
20 x$=left$(a$,6)
30 print x$
run
Teresa
```

Se *n*=0, x$ e una stringa nulla (" ")
Se *n*>len(a$), x$=a$

LEN — Funzione — 20, 16, 64, 128

Sintassi:
x=LEN(*a*$)

Abbreviazione:
nessuna

Restituisce la lunghezza (numero di byte) di una stringa.

```
10 a$="Teresa e' bella"
20 print len(a$)
run
15
```

LET **Istruzione** **20, 16, 64, 128**

Sintassi:
LET *variabile=espressione*

Abbreviazione:
lE

Nessun uso particolare. In altre parole, è sottinteso...; si può usare, ma come GOTO dopo THEN, non è necessario. Si può considerare come il *nome* dell'operazione fondamentale che assegna un valore a una variabile.

```
LET X=1   e   X=1
```

sono la stessa cosa.
Le variabili sono dei seguenti tipi:
— numerica (floating point o virgola mobile). Una lettera oppure una lettera seguita da un'altra lettera o da un numero. Esempi di variabili numeriche in virgola mobile:

```
A Al  AB PREZZO
```

L'ultima, poiché il calcolatore dà importanza solo alle prime due lettere, è esattamente uguale a PR... ma se non abbiamo problemi di memoria, scriverla per intero può essere utile per ricordare che cosa vuole dire. Può contenere fino a 255 caratteri, ma solo i primi due contano per distinguerla dalle altre.
– numerica intera (integer). Come sopra, seguita dal segno %. Non può naturalmente contenere una frazione decimale. Se scriviamo

```
x=10.5: x2%=x
```

il calcolatore non dà un messaggio d'errore: tronca x, e x2% assume il valore 10. Ciò può anche essere comodo. Inoltre le intere richiedono solo 2 byte ciascuna, contro i 5 delle variabili in virgola mobile (importante per le matrici grandi: si veda la voce DIM).
Un'altra limitazione è che il valore di x2% deve essere compreso tra x2% = +32767 e x2% = −32768.

Oltre a risparmiare memoria, l'uso di variabili intere consente una maggiore velocità di calcolo. Se un programma richiede una certa velocità sarà buona norma indicare come variabili intere tutte quelle che non devono superare i limiti accennati, in tutti i casi in cui non contano le frazioni.
— stringa. Il terzo tipo di variabile, a differenza dalle altre due, può contenere numeri e/o lettere, ma vengono tutti considerati "letteralmente", cioè non hanno valore numerico (si veda però la voce VAL).
Una stringa può essere lunga fino a 255 caratteri. Poiché INPUT accetta stringhe fino 160 caratteri, quelle più lunghe vengono create sommando due stringhe già esistenti (X$=A$+B$). Una stringa deve essere racchiusa fra virgolette, pena la comparsa del messaggio TYPE MISMATCH ERROR (incompatibilità del tipo).

LIST **Comando** **20, 16, 64, 128**

Sintassi:
LIST
oppure LIST *n*
oppure LIST *n1*–*n2*
oppure LIST –*n*
oppure LIST *n*–

Abbreviazione:
lI

Lista un programma intero, una riga, le righe tra *n1* e *n2*, le righe fino a *n* oppure le righe da *n* in poi. Il C-128 offre due modi di leggere il listato sullo schermo: premendo COMMODORE, la velocità dello "scroll" si rallenta; con NO SCROLL, si ferma del tutto. Si può ottenere lo stesso effetto premendo CTRL-S. In entrambi i casi, la lista torna a muoversi battendo un tasto qualsiasi. Per curiosità: si può listare un programma senza numeri di riga; si veda la voce POKE. Purtroppo, se una riga di programma contiene il comando LIST (come in molti programmi sul Lato 1 del dischetto),

```
100 list 120–130
```

RENUMBER non aggiorna i numeri di riga.
Normalmente un compilatore non accetta LIST in un programma, perché nel programma in p-code o linguaggio macchina che verrà prodotto, non esistono numeri di linea. Si consiglia perciò di sostituire LIST con un PRINT contenente il testo delle righe da riprodurre.

LOAD **Comando** **20, 16, 64, 128**

Sintassi:
LOAD ["*nomefile*"] [,D*n*] [,*ril*]

Abbreviazione:
lO

Carica un programma da disco o da nastro. LOAD da solo carica il primo programma esistente sul nastro. LOAD "*nomefile*" cerca sul nastro il programma e poi lo carica. LOAD "*nomefile*",8 è uguale a DLOAD"*nomefile*", cioè carica da disco.
$n=1$ carica da cassetta: poiché 1 è il default, può essere omesso se non si deve specificare il parametro *,ril*. I parametri da $n=8$ a $n=11$ caricano da uno dei 4 disk drive che possono essere presenti.
[,*ril*] serve soprattutto per i programmi in linguaggio macchina, che vengono locati in posizioni di memoria particolari. 0 (che equivale a non indicare niente, significa caricare il programma a partire dall'inizio dell'area del BASIC. 1, invece, significa "caricare il file nella stessa area di memoria in cui si trovava al momento in cui venne salvato").
L'avviamento del disco è naturalmente automatico. Per il nastro, se non è premuto il tasto PLAY, appare il messaggio PRESS PLAY ON TAPE. In un programma, LOAD "prog2" carica ed esegue "prog2", come descritto anche alla voce DLOAD.
Con il C-128 LOAD è indispensabile soltanto quando si vuole usare la cassetta: DLOAD, RUN e BLOAD, che sono più semplici nell'uso, lo possono sempre sostituire per il disco.

LOCATE **Istruzione** **16, 128**

Sintassi:
LOCATE *x, y*

Abbreviazione:
loC

Posiziona il pixel cursor (l'equivalente del cursore quando si opera in modo grafico) alle coordinate *x, y* indicate. I valori *x* e *y* possono essere contenuti in variabili e sono influenzati da SCALE.

```
10 locate 100,100
20 draw to 200,100
```

è la stessa cosa (ma può essere più comodo) che scrivere:

```
10 draw 1, 100,100 to 200,100
```

Si vedano anche la funzione RDOT e il programma JOYDRAW.

LOG — Funzione — 20, 16, 64, 128

Sintassi:
x=LOG(*n*)

Abbreviazione:
nessuna

Dà il logaritmo naturale (base e di *n* con $n>0$). Per avere il logaritmo base 10, dividere per log(10):

$\log_{10} n = \log(n)/\log(10)$

LOOP — Istruzione — 16, 128

Vedi DO.

MID$ — Funzione/Istruzione — 20, 16, 64, 128

Sintassi:
x$=MID$(*stringa,iniz,n*)

Abbreviazione:
mI

Questa funzione è sempre stata fondamentale nel BASIC. Mentre LEFT$ e RIGHT$ permettono di leggere gli *n* byte a sinistra o a destra in una stringa, MID$ permette di leggere un gruppo di *n* byte qualsiasi, e perciò le altre due funzioni non sono mai necessarie, sebbene possano essere più comode.
MID$ sul C-128 è sempre utile, anche se INSTR è più efficace per trovare un'espressione regolare all'interno di una stringa. Il C-128 offre una novità: MID$ è anche un'istruzione; può essere usata per modificare una stringa, come nei BASIC del PC.

Come *funzione*: estrae da *stringa* gli *n* caratteri a partire da *,iniz*:

```
10 A$="La Teresa e' bella"
20 x$=mid$(a$,4,6)
30 print x$
```

Il risultato del programma sarà la parola "Teresa".
"Teresa" è infatti la stringa che comincia nel quarto byte e finisce 6 byte più avanti. Un esempio è contenuto nel programma ORA ESATTA.

Come *istruzione*: (solo C-128) MID$ consente la sostituzione di uno o più caratteri in una stringa.

```
A$="LA TERESA E' BELLA"
MID$(A$,4,6)="FRANCA"
PRINT A$
LA FRANCA E' BELLA
```

Purtroppo, non è possibile allungare o raccorciare una stringa in questo modo; si otterrebbero risultati come:

```
LA MARILEN E' BELLA
LA ANNASA  E' BELLA
```

Nonostante questo, dà varie possibilità, di cui forse la seguente è la più importante.
Supponiamo di avere un lungo file in cui abbiamo usato un carattere (nell'esempio #) che ora vogliamo sostituire con un altro (*):

```
10 for t=1 to lf: rem lf=lung. file seq in x$(t)
20 :    x=instr(x$(t),"#")
30 :    if x then mid$(x$(t),x,1)="*": goto 20
40 next
```

Ovviamente si può sostituire qualsiasi gruppo di caratteri con un altro, purché i due gruppi abbiano la stessa lunghezza. Se non usiamo MID$, la routine dovrebbe avere una forma del tipo:

```
30 :    if x=0 then 40
35 :    x$(t)=left$(x$(t),x-1)+"*"+right$(x$(t),len(x$(t))-x): goto 20
```

In realtà, è possibile *ridurre* i caratteri visibili in una stringa, ma come mostra il prossimo esempio, la lunghezza della stringa rimane inalterata:

```
A$="1234567890": U$=CHR$(0)+CHR$(0)+CHR$(0)
MID$(A$,5,3)=U$: ? A$,LEN(A$)
1234890    10
```

È necessario CHR$(0): la forma alternativa "" (stringa nulla) non funziona. Questo sistema funziona bene: il risultato, sullo schermo o sulla stampante, è quello desiderato. In memoria resta qualche byte inutilmente occupato.

MONITOR — Istruzione — 16, 128

Sintassi:
MONITOR

Abbreviazione:
moN

Entra in modo MONITOR, cioè in quel settore del sistema operativo che opera direttamente in linguaggio macchina. L'uso del Monitor non è discusso in questo libro: certe condizioni di errore possono però lasciare il calcolatore in questo modo di operazione. Per uscirne battere x seguito da RETURN.

MOVSPR — Istruzione — 128

Sintassi:
MOVSPR*n*, *x1*, *y1*
oppure MOVSPR*n*, +/−*n*, +/−*n*
oppure MOVSPR*n*, *distanza*; *angolo*
oppure MOVSPR*n*, *angolo*#*velocità*

Abbreviazione:
mO

Fondamentale nell'uso degli sprite sul C-128. Posiziona o muove in vari modi e a varie velocità lo sprite *n*. La prima sintassi lo posiziona alle coordinate *x*, *y* (influenzate da SCALE); la seconda lo muove di +/−*n* pixel in direzione *x* e/o *y* rispetto alla posizione precedente; la terza lo muo-

ve di distanza *d* all'angolo *a* rispetto al punto di partenza, e la quarta lo muove all'angolo *a* alla velocità *#v* (0-15).
Questo libro tratta più ampiamente l'istruzione nei due capitoli dedicati agli sprite e nel programma tutorial SPRTUT.

NEW — Comando — 20, 16, 64, 128

Sintassi:
NEW

Abbreviazione:
nessuna

Elimina tutto, programma e variabili. Può essere usato anche in un programma, ed è pericoloso se il programma non è già stato salvato, ma può essere utile in certi casi (per esempio come una primitiva protezione per evitare la lettura del programma da parte di occhi estranei).
Non equivale esattamente all'uso del tasto RESET o allo spegnimento e riaccensione del calcolatore. Per esempio, se i tasti funzione sono stati riprogrammati, mantengono gli stessi valori (utile qualche volta). Lo schermo grafico non viene ripulito. Vedi anche CLR.

NEXT — Istruzione — 20, 16, 64, 128

Sintassi:
NEXT [*variabile, variabile...*]

Abbreviazione:
nE

Conclude il loop iniziato con un'istruzione FOR. Si veda la voce FOR. Qui osserviamo che quando abbiamo più loop "annidati" (l'uno dentro l'altro), l'ordine di chiusura è inverso. Cioè, il primo a essere aperto sarà l'ultimo a essere chiuso, e viceversa.

```
10 for x=1 to 3
20 :     for y=1 to 5
30 :         for z=1 to 10
40 :             print "x=",x
50 :             print "y=",y
60 :             print "z=",z
70 : next z, y, x
```

La riga 70 è uguale a NEXT Z: NEXT Y: NEXT X. Si può usare questa forma quando più loop si chiudono allo stesso momento.
Basta osservare attentamente l'esempio per capire che cosa succederebbe se chiudessero i loop nell'ordine sbagliato.
Se invece un solo loop è attivo, NEXT da solo lo chiuderà:

```
10 for t=1 to 3: print t: next
```

NOT — Operatore logico — 20, 16, 64, 128

Sintassi:
IF NOT B THEN ...
oppure NOT X

Dà il complemento logico (complemento a 1). In termini di operazioni binarie, la tabella è:

X	NOT X
1	0
0	−1

In altre parole NOT X= −(X+1). Si vedano anche le voci AND, OR, XOR.

ON... GOTO/GOSUB — Istruzione — 20, 16, 64, 128

Sintassi:
ON *espressione* GOTO *n1, n2, n3...*
oppure ON *espressione* GOSUB *n1, n2, n3...*

Abbreviazione:
ON... gO / ON... goS

Forse usata troppo raramente, questa istruzione è molto potente. Permette al programma di scegliere tra un elenco di numeri di riga diversi (*n1, n2...*), in genere dopo un'istruzione INPUT. Se la variabile ha valore 1, il programma va a *n1*, se 2, a *n2* ... e così via.

```
ON X GOTO 100, 200, 230, 10
```

è come scrivere:

```
IF X=1 THEN 100
IF X=2 THEN 200
```

```
IF X=3 THEN 230
IF X=4 THEN 10
```

Se X fosse inferiore a 1 o superiore al numero di righe specificate, il programma passerebbe alla riga immediatamente successiva a quella contenente l'istruzione ON. Se si usa GOSUB, il RETURN riporta alla riga successiva.
I vari menu sul disco consentono di caricare un programma battendo una sola lettera. Il meccanismo è dato da

```
100 getkey x$
110 x=asc(x$)-64
120 on x goto 200,300,400...
```

Poiché la A ha codice ASCII 65, se X$="A", ASC(X$)-64 dà 1, e così via. Fin qui, esiste la limitazione dovuta al fatto che l'espressione (X o Y+3 negli esempi) deve assumere i valori 1,2,3,... Ma si può fare in modo che sia sempre così.
Se infatti l'espressione Y+3=1 è vera, ON Y+3=1 assumerà per l'operatore logico il valore di −1. Ciò non andrebbe bene perché ON seguito da un valore negativo darebbe ILLEGAL QUANTITY ERROR e il programma si fermerebbe. Ma

```
ON-(Y+3=1)
```

invertirebbe il segno, e sarebbe come scrivere ON 1. La parte in parentesi avrà sempre il valore −1 se è vera e 0 se è falsa. Si può quindi scrivere una serie di queste espressioni moltiplicando la prima per −1, la seconda per −2, e così via.
Le seguenti due righe in BASIC daranno lo stesso risultato:

```
IF X=7 THEN 400: ELSE IF X=8 THEN 200: ELSE IF X>8 THEN 300
ELSE IF X<7 THEN 1200
ON-(X=7) -2*(X=8) -3*(X>8) -4*(X<7) GOTO 400, 200, 300, 1200
```

Ciascuna delle espressioni tra parentesi avrà sempre il valore 0 oppure −1. Moltiplicandoli per −1, −2, −3, −4... avranno quindi sempre il valore 0 oppure +1, +2, +3, +4... che è ciò che desideriamo. Usare un operatore logico così richiede una certa chiarezza mentale, ma una volta che si capisce ciò che si sta facendo, il programma risulta più snello e più efficace.

OPEN **Istruzione** **16, 64, 128**

Sintassi:
OPEN *lfn, dn* [*indirizzo secondario*] [,"*nomefile,tipo, modo*"] [,*cmd*$]

Abbreviazione:
oP

Apre un file per leggere/ricevere dati o per scrivere/trasmettere. Tutte le forme di input/output, compreso lo schermo, possono essere aperte e chiuse come file.
lfn è il numero (1 – 255) del file da aprire
dn è il numero del dispositivo:
0 = tastiera
1 = registratore a cassetta
2 = interfaccia RS232
3 = schermo
4-7 = stampanti
8-11 = disk drive

indirizzo secondario varia, quando serve, a seconda del dispositivo:
Cassetta:

0 = lettura
1 = scrittura
2 = scrittura con carattere di fine nastro (EOT)

Disco:

1-14 = canali per dati
15 = canale dei comandi

Stampante:

0 = modalità maiuscolo/caratteri grafici
7 = modalità maiuscolo/minuscolo

RS232

La System Guide non dà informazioni precise per l'interfaccia RS232. In genere chi acquista un modem o stampante che la richiede troverà le istruzioni necessarie insieme all'apparecchio.

,tipo indica il tipo di file: programma, sequenziale, relativo, utente
,modo lettura (R), scrittura (W)

Esempi:

OPEN 1,1,0, "file"	legge da nastro
OPEN 1,1,1, "file"	scrive su nastro
OPEN 1,1,2, "file"	scrive su nastro con EOT
OPEN 1,2,0, "stringa"	apre canale a RS232
OPEN 1,3	apre lo schermo come file
OPEN 1,4,0, "stringa"	manda maiuscole e caratteri grafici alla stampante (seguirà un PRINT#).
OPEN 1,4,7	manda maiuscole e minuscole alla stampante (seguirà un PRINT#1)

Ogni file che viene aperto deve essere chiuso con il comando CLOSE, ma la tastiera è normalmente aperta. OPEN è necessario prima delle istruzioni: CMD, GET#, INPUT#, PRINT#, pena il messaggio FILE NOT OPEN ERROR. Si vedano le voci DOPEN, CLOSE e DCLOSE.

OR — Operatore logico — 20, 16, 64, 128

Sintassi:
x=1 OR 2
oppure IF x=2 OR y=7 THEN ...

In BASIC il significato è OPPURE, per cui nell'esempio dato nella sintassi il programma passerà alla riga successiva se $x<>2$ e $y<>7$.
La tabella della verità di OR per le operazioni binarie è:

1 OR 1=1   1 OR 0=1
0 OR 1=1   0 OR 0=0

Confrontare questo con la tabella per l'OR-esclusivo (XOR). Mentre per OR il risultato è 1 se uno o entrambi i membri sono veri, per XOR 1 è dato da uno, ma non entrambi. Si veda anche il programma ORANDXOR (Lato 1), oltre alle voci AND e XOR.

PAINT — Comando — 16, 128

Sintassi:
PAINT [*fonte colore*],*x*,*y* [,*modo*]

Abbreviazione:
pA

Dipinge una zona dello schermo nel *fonte colore* selezionato.

fonte colore va da 0 a 3: si veda la spiegazione e la tabella di colori alla voce COLOR.

x,y le coordinate (all'interno della zona da colorare) da cui parte l'operazione (il default è la posizione del pixel cursor)

modo 0=dipinge fino a un bordo nello stesso colore del *fonte colore* indicato (è il default). 1=dipinge fino a qualsiasi bordo. Con *modo* zero e *fonte colore* (per esempio) blu, supponiamo di aver disegnato con una linea blu un quadrato e, all'interno di questo, un cerchio con bordo rosso. Se ora dipingiamo con un *fonte colore* che in questo momento è blu, a partire dal centro del cerchio, il cerchio scomparirà e tutto il quadrato si riempirà di blu. Se invece il *modo* è 1, solo il cerchio si riempirà di blu. Il programma PAINT illustra questo meglio di molte parole.

PEEK — Funzione — 20, 16, 64, 128

Sintassi:

v=PEEK(*n*)

Abbreviazione:
peE

Legge il valore *v* (da 0 a 255) contenuto nella posizione di memoria *n*. Anche se mantiene la stessa sintassi che ha su tutti i calcolatori a partire dal VIC-20, e si può usare senza la variabile *v* indicata nella sintassi:

```
10 print peek (4096)
```

si deve tenere presente che il C-128 ha diversi banchi di memoria. Perciò quest'ultimo esempio darà il valore contenuto nella locazione 4096 del banco 15 (il default): se vogliamo leggere un altro banco, dovremo prima selezionarlo con il comando BANK. Si vedano anche la voce POKE e il programma PEEK/POKE.

PEN — Funzione — 128

Sintassi:
x=PEN(*n*)

Abbreviazione:
pE

Restituisce la posizione (coordinate *x, y,* non influenzate da SCALE) della penna luminosa.

n=0 oppure 2 rispettivamente per la coordinata *x* in 40 e 80 colonne
n=1 oppure 3 per la coordinata *y* (40/80)
n=4 dà il valore del trigger.

PLAY **Istruzione** **128**

Sintassi:
PLAY "[*Voce*] [*Ottava*] [T(*envelope*)] [U(*volume*)] [X(*filtro*)] [*note*]
Abbreviazione:
pL

Suona una o più note musicali espresse in forma di stringa. Le note si specificano con CDEFGAB (Do Re Mi Fa Sol La Si), la Voce con V, l'Ottava con O, l'inviluppo (envelope) con T, il volume con U, il filtro (FILTER) con X.
Questa voce è più ampiamente illustrata nel capitolo PLAY e dai programmi del Menu Musicale.

POINTER **Funzione** **128**

Sintassi:
a=POINTER(*var*)

Abbreviazione:
poI (non è indicata nell'Appendice K della System Guide)

Restituisce l'indirizzo in cui è conservato il valore di una variabile.

POKE **Istruzione** **20, 16, 64, 128**

Sintassi:
POKE *indirizzo, valore*

Abbreviazione:
poK

Cambia il valore contenuto in una posizione di memoria: ha la stessa sintassi per tutti i calcolatori, ma per il C-128 bisogna tener presente che se vogliamo inserire i valori in un banco di memoria diverso dal 15, bisogna usare l'istruzione BANK prima del primo POKE della serie. Supponiamo di aver già fatto dei POKE in un banco diverso dal 15 (che è il default). Vogliamo inserire il valore 1 nell'indirizzo 1024 — ciò farà comparire la lettera A in colonna 0, riga 0 dello schermo a 40 colonne (1024 è infatti il primo indirizzo della memoria per lo schermo a 40 colonne — l'ultimo è 2023).
La riga 10 serve nell'ipotesi di aver usato in precedenza un banco diverso dal default (15):

```
10 bank 15
20 poke 1024,1
```

La riga 30 usa un loop:

```
30 for t=1024 to 1024+26
40 :    poke t, a-1023
50 next
```

Si veda anche il programma PEEK/POKE sul dischetto.
Infine, alcuni POKE utili:

POKE 775, 139	disabilita	LIST	POKE 775, 81	riabilita	LIST
POKE 818, 180	»	SAVE	POKE 818, 78	»	SAVE
POKE 816,0	»	LOAD	POKE 816,108	»	LOAD
POKE 808,100	»	RUN/STOP	POKE 808,110	»	RUN/STOP
POKE 792,125	»	RESTORE	POKE 792,64	»	RESTORE
POKE 2592,0	»	tastiera	POKE 2592,10	»	tastiera
POKE 2594,94	»	ripetiz. tasti	POKE 2594,255	»	ripetiz. tasti

POKE 24,37 il LIST omette i numeri di riga
POKE 24,27 riabilita la numerazione delle righe.

POS　　　　funzione　　　　20, 16, 64, 128

Sintassi:
x=POS(n)

Abbreviazione:
nessuna

Dà la colonna in cui si trova attualmente il cursore. L'argomento n è un "dummy". Qualsiasi valore va bene.

POT **Funzione** **128**

Sintassi:
x=POT(n)

Abbreviazione:
pO

Dà il valore (da 0 a 255) del potenziometro (paddle), con n=numero del potenziometro (da 1 a 4). Se x=>256 significa che è premuto il tasto FIRE. Se non è collegato nessun potenziometro il valore è 255.

PRINT **Istruzione** **20, 16, 64, 128**

Sintassi:
PRINT "*caratteri*"
oppure PRINT *variabile stringa*
oppure PRINT *variabile numerica*
oppure PRINT *espressione/funzione*
oppure PRINT *interpunzioni* [, ;]

Abbreviazione:
?

Scrive qualche cosa sullo schermo (PRINT) oppure su file (PRINT#).

? "caratteri"	stampa esattamente ciò che si trova tra le virgolette
? x$	stampa la stringa contenuta in x$
? x	stampa il valore numerico contenuto in x
? x*3	stampa il risultato di questa (o qualsiasi altra) operazione
? peek (4096)	stampa la funzione PEEK relativa alla locazione 4096
? a$, b$	stampa le due stringhe separate da un numero variabile di spazi (la virgola fa un salto fino alla colonna 9, 19, 29... a partire dalla posizione del cursore)
? a$;	stampa a$ senza andare a capo: così il PRINT successivo continuerà sulla stessa riga
?	stampa un CHR$(13): cioè va alla colonna zero della riga successiva.

All'interno delle virgolette, molti caratteri hanno funzioni speciali e generano speciali simboli sullo schermo, che poi nell'esecuzione del programma producono effetti particolari. Per vedere gli effetti, scrivere:

```
10 ?" ...
```

e dopo le virgolette battere i tasti 1 2 3 4 5 6 7 8 9 0 tenendo premuto il tasto CONTROL, poi ripetere tenendo premuto COMMODORE. Le frecce (senza CONTROL) producono 4 simboli, come pure HOME e CLR. Ciascuno di questi simboli produrrà, nel programma, lo stesso effetto che si otterrebbe con gli stessi tasti in modo diretto. CLR pulirà lo schermo; CONTROL-3 colorerà i caratteri in rosso: freccia verso l'alto sposterà il cursore di una riga in su, e così via. Si vedano anche le istruzioni PRINT#, PRINT USING, CHAR, e le funzioni TAB, SPC, POS.

PRINT# **Istruzione** **20, 16, 64, 128**

Sintassi:
PRINT# *n*, *caratteri/variabili*

Abbreviazione:
pR (NON usare ?#)

Manda dati al file #*n*, che deve essere stato aperto con una istruzione OPEN. Il file può essere diretto non solo verso il disco o nastro, ma anche verso stampante, linea telefonica, ecc. Si veda OPEN.

OPEN 7,4

Una volta che è stato aperto un canale (4=stampante) insieme a un file (#7 in questo caso), PRINT#7, seguito da caratteri tra virgolette, variabili stringa, variabili o espressioni numeriche, produce circa gli stessi risultati di PRINT. Non si può usare CHAR; TAB e SPC possono non dare i risultati desiderati. Non è ammesso uno spazio tra PRINT e #.
PRINT# serve anche a mandare comandi a periferiche (stampante, drive). Tutte le funzioni speciali della stampante possono essere ottenute mandando alla stampante una serie di caratteri speciali. Alcuni di questi sono caratteri ASCII standard (CHR$(13)=RETURN; CHR$(12)=avanzamento modulo, per esempio). Altre funzioni sono ottenute inviando prima il CHR$(27), (ESCAPE), seguito poi da una serie di CHR$(*n*), che assumono così significati speciali. Per esempio:

PRINT#7, CHR$(27);CHR$(69)

dice alla stampante DPS 1101 di cominciare a sottolineare automaticamente le parole. Il CHR$(69) è normalmente la lettera E, e infatti si può anche scrivere:

PRINT#7, CHR$(27);"E"

Analogamente, PRINT# serve per mandare istruzioni al drive lungo il canale 15:

OPEN15,8,15: PRINT#15,"INITIALIZE"

Si tenga presente che si può usare CMD per evitare di usare PRINT# (in particolare per listare un programma con la stampante o su disco (o nastro): OPEN 4,4

```
CMD4,"LISTA"
LIST
PRINT#4
CLOSE4
```

Il PRINT# qui è necessario per ridirigere l'output verso lo schermo. Mette semplicemente fine alla modalità CMD.

PRINT USING e PRINT# USING **Istruzioni** **16, 128**

Sintassi:
PRINT USING "*formato*"; *testo...*
oppure PRINT#*n*, USING "*formato*"; *testo...*

Abbreviazione:
? usI, pR usI [anche senza spazi]

Utilizza una stringa "*formato*" che definisce il modo di interpretare e posizionare stringhe o valori numerici per organizzare la stampa di documenti complessi su schermo, stampante, disco, ecc. L'istruzione è trattata nel Capitolo 5 di questo libro e nel programma omonimo: qui riassumiamo solo i simboli da usare:

#	riserva spazio per un carattere
+	ordina di stampare + o − davanti a numeri positivi o negativi. Può apparire all'inizio o alla fine della stringa *formato*, ma non in entrambe le posizioni.
−	come +, ma se un numero è positivo stampa uno spazio invece del +.
.	stabilisce la posizione del punto decimale, e di conseguenza la lunghezza della frazione dopo il punto. Per esempio: "####.##" troncherà i decimali a due cifre, arrotondando l'ultima in basso se =<5, in alto se >5.
,	permette l'uso della virgola (in modo anglosassone, per indicare migliaia). Ma usando PUDEF si può ancora invertire l'uso e stam-

pare all'italiana, con la virgola per i decimali e il punto (o lo spazio o quel che si vuole) per le migliaia.

$ fa precedere il segno $ al numero. Anche in questo caso PUDEF può ridefinire $ (come il segno per la Lira, per esempio).

^^^^ indica di esprimere i numeri con notazione scientifica. Il simbolo ^ corrisponde alla pressione del tasto immediatamente a sinistra del tasto RESTORE.

= centra una stringa nel campo riservatole.

> allinea a destra una stringa. I numeri vengono sempre allineati a destra, naturalmente.

PUDEF **Comando** **16, 128**

Sintassi:
PUDEF"*nnnn*"

Abbreviazione:
pU

Ridefinisce i 4 caratteri spazio, virgola, punto e $ per l'istruzione PRINT USING. Per esempio:

PUDEF"*.,L"

Data questa nuova definizione, PRINT USING stamperà (fino a nuovo ordine)* al posto dello spazio, virgole al posto dei punti, punti al posto delle virgole e L al posto di $.
In altre parole, la prima posizione è sempre lo spazio, la seconda la virgola, e così via. La stringa "*nnnn*" deve sempre definire ciascuna posizione.
Se vogliamo ridefinire solo $, quindi, dobbiamo scrivere

PUDEF " ,.L"

riconfermando cioè i valori di default per i primi tre. Se volessimo invece ridefinire solo la virgola (come spazio nell'esempio seguente), basterà:

PUDEF " "

Infatti i due caratteri a destra della virgola non sono chiamati in causa e restano " .$".
PUDEF influenza solo le espressioni numeriche: le stringhe (ed è logico che sia così) non vengono modificate.
PUDEF è più ampiamente trattato nel Capitolo 5, nonché nel programma PUDEF.

RCLR **Funzione** **16, 128**

Sintassi:
x=RCLR(n)

Abbreviazione:
rC

Restituisce un numero (1-16) corrispondente al colore attuale della fonte colore n. Per i valori di n si veda la voce COLOR.
PRINT RCLR(5) dà il colore del carattere (40 e 80 colonne)
PRINT RCLR(4) dà il colore del bordo (40 colonne)
PRINT RCLR(6) dà il colore dello sfondo (80 colonne).
Si noti che RCLR(5) può avere due valori diversi simultaneamente per i due schermi. Se sullo schermo a 40 colonne il carattere è nero verrà stampato 1, se invece passiamo allo schermo a 80 colonne, e il carattere non è nero, verrà stampato un numero diverso.

RDOT **Funzione** **16, 128**

Sintassi:
v=RDOT(n)

Abbreviazione:
rD

Restituisce le coordinate o il colore del pixel cursor. Si veda anche LOCATE.

n=0 coordinata x
n=1 coordinata y
n=2 colore (1-16)

READ **Istruzione** **20, 16, 64, 128**

Sintassi:
READ *variabile* [,*variabile*]...

Abbreviazione:
reA

Durante l'esecuzione di un programma legge dati nelle istruzioni DATA e li carica in memoria nella variabile o nelle variabili indicate. Si vedano anche

le voci DATA e RESTORE, nonché il programma RESTORE.
Può caricare un qualsiasi numero di dati in un qualsiasi numero di variabili (o matrici).

```
10 read A$m a%, a
100 data parola, 99, 33.333
```

Se immaginiamo le istruzioni DATA come molto più numerose, resta inteso che, perché la riga 10 funzioni sempre, è necessario che la prima, quarta, settima... istruzioni di DATA siano stringhe.
Si possono anche caricare DATA in matrici:

```
10 dim a(5)
20 for t=0 to 4
30 :   read a(t)
40 next t
100 data 10, 11, 99, 999, 0
```

RECORD   **Istruzione**   **128**

Sintassi:
RECORD# *n*, *record* [,*byte*]

Abbreviazione:
rE

Posiziona un puntatore in un file *n*, al *record* indicato su un determinato *byte*.
Il Manuale d'Uso del 1571 fornisce molte informazioni chiare ed utili sui file relativi in generale, e un ottimo esempio sull'uso di RECORD. L'esempio dato nella System Guide alla pagina 17-62 apre un file relativo con

```
10 dopen#2, "customer"
20 record#2,10,1
30 print#2, a$
40 dclose#2
```

RECORD qui posiziona il puntatore sul record 10 al byte 1: la 30 scrive a$ nel file.
Se non esistesse ancora il record 10, questo verrebbe creato (naturalmente insieme a tutti quelli intercorrenti tra il numero più alto di record già esistente e il 10).

REM — Istruzione — 20, 16, 64, 128

Sintassi:
REM

Abbreviazione:
nessuna

Utilissimo anche se non esegue nessuna operazione. Serve infatti per inserire in un programma un promemoria (reminder) di qualsiasi genere:

```
35 rem inserire qui una routine
45 rem eliminare le righe 50-90
95 rem inizio programma principale
105 rem gosub 2000
```

La riga 105 serve per sospendere temporaneamente l'esecuzione della subroutine 2000, per esempio perché non esiste ancora o perché esiste e funziona male; o anche semplicemente perché funziona bene e non vogliamo perdere tempo ogni volta che collaudiamo il resto del programma.
È meglio non usare maiuscole nelle REM: infatti il BASIC le interpreta come caratteri speciali e le sostituisce con comandi BASIC (che non vengono eseguiti, ma che non aiutano certo a capire ciò che si voleva scrivere). La stessa cosa accade con le istruzioni DATA se si inseriscono stringhe con lettere maiuscole senza racchiuderle tra virgolette. In questo caso i risultati del programma sono quelli desiderati, ma il comando LIST produce stranezze come "rclaria" al posto di "Maria". Si veda anche la voce DATA.

RENAME — Comando — 16, 128

Sintassi:
RENAME [D*n*]"*vecchionome*" TO "*nuovonome*"[U*n*]

Abbreviazione:
reN

Cambia il nome di un file su disco. Non riscrive il file: si limita ad aggiornare il direttorio. Il file non deve essere aperto, e *nuovonome* non deve essere il nome di un file già sul disco.
Si può usare in modo diretto:

```
RENAME "INFERNO" TO "PARADISO", U9
```

oppure da programma; in quest'ultimo caso può essere utile servirsi di variabili stringa per i nomi dei file. Queste devono essere poste tra parentesi:

```
1000 rename (n1$) to (x$)
```

RENUMBER **Comando** **16, 128**

Sintassi:
RENUMBER *nn*
oppure RENUMBER *nn, ni,*
oppure RENUMBER *nn, ni, nv*

Abbreviazione:
renU

Rinumera con l'incremento *ni* a partire dal nuovo numero di partenza *nn*; rinumera le righe a partire dalla vecchia riga *nv*, dando a queste numeri nuovi a partire da *nn* e con l'incremento *ni*

```
RENUMBER    oppure
RENUMBER 10 oppure
RENUMBER 10, 10
```

rinumerano tutto il programma a partire dalla riga 10 e con intervalli di 10 fra le righe. I default di *nn, ni* sono infatti 10, 10.
RENUMBER 50, 15 rinumera tutto il programma con intervalli di 15 a partire dal nuovo numero iniziale di 50.
RENUMBER 100,10, 90 rinumera le righe da 90 in poi con intervalli di 10: la vecchia 90 diventa 100 e le righe prima di 90 non cambiano.
Il comando è ulteriormente spiegato nel Capitolo 4 di questo libro.

Esempio:
programmino prima e dopo RENUMBER

```
10 print "test"
20 gosub 43: list 43
30 stop
43 print "fine"
44 return

RENUMBER 100, 10, 43
10 print "test"
```

```
20 gosub 100: list 43
30 stop
100 print "fine"
110 return
```

RENUMBER aggiorna i rimandi nei GOTO, GOSUB, TRAP, ecc. RENUMBER, purtroppo, in un caso non rinumera i rimandi. Come si vede nell'esempio, i comandi LIST nel programma restano con i numeri vecchi.

RESTORE **Istruzione** **20, 16, 64, 128**

Sintassi:
RESTORE [*riga*]

Abbreviazione:
reS

Sul VIC-20 e sul C-64 non si può specificare un numero di riga. Senza numero di riga RESTORE "azzera" la lettura delle istruzioni DATA. Ordina cioè al programma di tornare a leggere DATA a partire dall'inizio (la riga DATA che ha il numero più basso). Con un numero di riga, il programma va a leggere i DATA a partire da quella riga. Il numero di riga può essere contenuto in una variabile. Se quindi organizziamo le istruzioni DATA in modo veramente razionale, si hanno i vantaggi:

- Se una riga DATA (o un gruppo di righe DATA) è richiesta frequentemente, si può usare RESTORE per leggerla tutte le volte che si vuole.
- La riga indicata può essere contenuta in una variabile e variare a seconda degli eventi che hanno luogo durante lo svolgimento del programma. Si veda anche il programma RESTORE.

```
restore.list
5 data "Animali"
10 read a$
20 print "s";a$
30 input "Ti piacciono cani o gatti";x$
40 if x$ = "cani" then restore 1000:
   else if x$ = "gatti" then restore 2000: else 40
50 read a$
60 print a$; a$
70 print "Quanti ";x$;" vorresti (1-2)";:
   input n: if n>2 then 70
80 restore n*100
85 read a$
```

```
90 print a$
100 data  uno basta.
200 data  due sono compagnia
1000 data  " bau"
2000 data  " miao"
```

Il lettore perdoni l'idiozia del programma, che si trova anche sul disco.

RESUME — Istruzione — 16, 128

Sintassi:
RESUME [*riga*|NEXT]

Abbreviazione:
resU

Dopo una TRAP il programma si trova alla riga specificata nella TRAP, oppure alla fine della routine che inizia a quella riga. Bisogna indicare il punto in cui deve proseguire.
RESUME da solo dice al programma di eseguire di nuovo la riga in cui si è verificato l'errore:

```
10 trap 100
20 input ......
99 end
100 print "errato: riprovare": resume
```

RESUME NEXT dice al programma di andare alla riga successiva a quella dell'errore

```
10 trap 100
20 input n: x=y/9
30 print "....."
99 end
100 ?"valore errato: pazienza, andiamo avanti": resume 30
```

In quest'ultimo esempio, per un errore in 20,

```
100 ?"valore errato: pazienza": resume next
```

avrà lo stesso effetto.
Si veda anche la voce TRAP. TRAP e RESUME sono inoltre discussi in un altro capitolo.

RETURN **Istruzione** **20, 16, 64, 128**

Sintassi:
RETURN

Abbreviazione:
reT

Alla fine di una subroutine RETURN dice al programma di tornare al punto di partenza e di riprendere a partire dall'istruzione immediatamente successiva al GOSUB. Si vedano le voci GOSUB e COLLISION.

RGR **Funzione** **16, 128**

Sintassi:
x=RGR(*n*)

Abbreviazione:
rG

Indica il modo grafico attuale. L'argomento *n* è un "dummy", cioè qualsiasi valore di *n* dà lo stesso risultato. Questa funzione diventa preziosa quando il calcolatore non sa se l'utente ha un monitor a 80 o a 40 colonne. All'inizio di un programma il sistema può trovarsi nelle seguenti condizioni:

- modo 40 colonne, monitor correttamente collegato
- modo 40, monitor collegato su 80
- modo 80, monitor correttamente collegato
- modo 80, monitor su 40.

Se il programma non è grafico, può funzionare sia a 80 che a 40 colonne; se è grafico, deve funzionare a 40 colonne. Sul disco che accompagna questo libro, il Menu Comandi Grafici deve in certi casi chiedere all'utente di usare lo schermo a 40 colonne, mentre programmi come 80-COL richiedono il contrario. Il menu stesso deve essere visibile in ogni caso. Usando un'espressione del tipo IF RGR(0)>4 THEN...: ELSE... si è sempre in grado di far comparire qualche cosa sullo schermo, qualunque sia il modo di operazione. Si suggerisce di esaminare il listato dei vari programmi.
Alle routine di questo genere si possono aggiungere istruzioni particolari a seconda del modo (80 o 40). L'espressione IF RGR(0)>4 va spiegata, specie perché la System Guide non spiega chiaramente i valori 6 e 8. PRINT RGR(0) può fornire uno di 7 valori:

0: schermo di testo a 40 colonne
1: schermo grafico alta risoluzione
2: schermo grafico con finestra sullo schermo a 40
3: schermo grafico a bassa risoluzione
4: schermo grafico a bassa risoluzione con finestra 40 colonne
5: schermo di testo a 80 colonne
6: schermo grafico 1 e testo ad 80 colonne
8: schermo grafico 3 e testo ad 80 colonne.

RIGHT$ — Funzione — 20, 16, 64, 128

Sintassi:
x$=RIGHT$(*stringa,n*)

Abbreviazione:
rI

Restituisce una stringa contenente gli *n* caratteri più a destra in *stringa*.

```
10 a$="La Teresa è bella"
20 x$=right$(a$,5)
30 print x$
RUN
bella
```

Si vedano anche le voci LEFT$ e MID$, nonché i capitoli dedicati alle stringhe.

RND — Funzione — 20, 16, 64, 128

Sintassi:
x=RND(*n*)

Abbreviazione:
rN

Dà un numero casuale maggiore di 0 e inferiore a 1.
$n=0$ dà un numero basato sull'orologio interno
$n>1$ da un numero pseudocasuale riproducibile, basato sul numero seme
$n<0$ produce una base numerica chiamata "seme".
All'inizio di un programma creare un seme, per esempio con *x*=RND(−TI), dove TI è l'orologio interno. In seguito usare RND(*x*). Poi-

ché il valore prodotto è una frazione <1, se vogliamo un numero a caso, per esempio nella gamma 0-100, bisogna usare un'espressione del tipo:

x=INT(RND(n)* 100)+1

L'uso della funzione INT serve per avere un numero intero.

RSPCOLOR **Funzione** **128**

Sintassi:
x=RSPCOLOR(n)

Abbreviazione:
rspC

Per n=1 ed n=2 rispettivamente, dà i valori multicolor vigenti per gli sprite (si veda la voce SPRCOLOR).

RSPPOS **Funzione** **128**

Sintassi:
x=RSPPOS(*n1*,*n2*)

Abbreviazione:
rS

Dà la posizione dello sprite indicato con *n1*.
n2 può avere i seguenti valori:

0 dà la coordinata x
1 dà la coordinata y
2 dà la velocità (0-15)

RSPRITE **Funzione** **128**

Sintassi:
x=RSPRITE(*n1*,*n2*)

Abbreviazione:
rspR

Dà i parametri dello sprite *n1*. Il parametro *n2* può avere i seguenti valori:

0 Acceso (1) o spento (2)
1 Colore fondamentale (1-16)
2 Priorità (0/1)
3 Espansione *x* (0/1)
4 Espansione *y* (0/1)
5 Multicolor (0/1)

Per ulteriori spiegazioni di questi parametri si vedano la voce SPRITE e i capitoli dedicati all'argomento. Attenzione ai numeri: SPRITE ha 7 parametri: il secondo parametro di SPRITE (acceso/spento) corrisponde al valore 0 di *n2* in RSPRITE.

RUN **Comando** **20, 16, 64, 128**

Sintassi:
RUN [*riga*]
oppure RUN "*nomeprog*"

Abbreviazione:
rU

Esegue un programma già in memoria, dall'inizio (RUN) oppure dalla *riga* specificata; oppure carica un programma e lo manda in esecuzione.

RUN "*nomeprog*"

carica il programma (esclusivamente da disco e in modo 128) e lo esegue. "*nomeprog*" può essere contenuto in una variabile, la quale deve essere inserita tra parentesi:

RUN (A$)

Può essere usato in un programma:

RUN 50 farà ripartire il programma dalla riga 50
RUN "nuovoprog" caricherà "nuovoprog" e lo farà partire (dalla prima riga).

Nota: RUN cancella le variabili presenti in memoria. RUN "nuovoprog" cancella anche il programma attualmente in memoria: è come scrivere

```
NEW
LOAD "NUOVOPROG"
RUN
```

In un programma, sia RUN che LOAD/DLOAD possono caricare ed eseguire un nuovo programma.

```
10 run "secondoprog"
```

carica "secondoprog" e lo fa partire "a freddo", cioè con tutte le variabili azzerate.

```
10 dload "secondoprog"
```

carica "secondoprog" e lo fa partire "a caldo", cioè con tutte le variabili ancora in memoria.
RUN è usato così in tutti i programmi sul dischetto che accompagna questo volume, almeno per ricaricare il programma menu. È necessario stare attenti mentre si scrive un programma che termina in questo modo, perché verrà eliminato dalla memoria: quindi un programma con RUN "nuovoprog" DEVE essere salvato prima di essere eseguito, a meno che non si tolga il dischetto dal drive: così si riceve il messaggio d'errore FILE NOT FOUND ERROR IN..., che nel caso conferma che la riga del programma è corretta.

RWINDOW **Funzione** **128**

Sintassi:
x=RWINDOW(*n*)

Abbreviazione:
rW

Dà i parametri della finestra attualmente definita. I valori di *n* sono:

0 numero di righe nella finestra
1 numero di colonne
2 formato (40 o 80 colonne)

Un errore di stampa (lines?/rows?) non rende più chiara la spiegazione nella System Guide. In pratica la definizione usa parametri del tutto simili a CHAR. Si veda anche WINDOW.

SAVE — Comando — 20, 16, 64, 128

Sintassi:
SAVE ["*nomefile*"] [,D*n*] [,EOT]

Abbreviazione:
sA

Salva (immagazzina) un programma presente nella memoria del calcolatore, su nastro o su disco.

SAVE "*nomefile*" lo salva su nastro col nome
SAVE "*nomefile*",1,1 lo salva su nastro negli stessi indirizzi in cui risiede attualmente.
SAVE "*nomefile*",1,2 lo salva con EOT (fine nastro, un carattere speciale)
SAVE "*nomefile*",1,3 lo salva negli stessi indirizzi con EOT
SAVE "*nomefile*",8 (o altro numero di unità disco) lo salva su disco.
SAVE "*@nomefile*",8 oppure DSAVE"*@nomefile*" o BSAVE "*@nomefile*":
dovrebbero salvare "*nomefile*" sostituendolo al programma omonimo precedente. La System Guide giustamente non ne parla, perché è sconsigliabile usare questa opzione: per un errore nel sistema operativo dei drive 1541, 1570 e 1571 è possibile danneggiare gravemente il direttorio, perdendo parte del file (o di un altro file già sul disco). Per non correre rischi, si deve usare SCRATCH "*nomefile*": SAVE/DSAVE "*nomefile*". Verificare il risultato con DVERIFY.

Si vedano anche le voci DSAVE, LOAD, DLOAD, BSAVE, BLOAD.

SCALE — Comando — 16, 128

Sintassi:
SCALE *n* [,*x* ,*y*]

Abbreviazione:
scA

Cambia la gamma di coordinate *x,y* sullo schermo grafico. Il primo numero *n* è ON/OFF (1/0).
Con SCALE 0, $x=319$ e $y=199$, oppure in modo multicolor, $x=159$ e $y=199$ (perché in multicolor ogni pixel è doppio sull'asse *x*).
Visto che il numero dei pixel disponibili è sempre lo stesso, perché cam-

biare i numeri con SCALE? Se stiamo tracciando un grafico basato su dati numerici (una curva per esempio) può essere comodo far combaciare i numeri di pixel con (per esempio) i valori in Lire (o la tensione in volt...). SCALE dà quindi la possibilità di evitare calcoli complessi e lenti. Con SCALE 1, invece,

$x^{max} > 319$ e < 32767 mentre $y^{max} > 199$ e < 32767

SCALE 1 senza numeri sceglie default 1023 per x e y in modo grafico normale, 2047, 1023 in multicolor.
SCALE torna ai default (319,199) ogni volta che viene dato un comando GRAPHIC: perciò è necessario ricordarsi di rinnovare SCALE se durante l'uso dello schermo grafico bisogna passare allo schermo normale e viceversa. In ogni caso, quindi, la forma sarà:

GRAPHIC 1: SCALE 1, 640, 200

Quest'ultima SCALE è utile da ricordarsi, perché, se scriviamo un programma che usa sia GRAPHIC 1/2 che GRAPHIC 3/4, questi valori daranno gli stessi risultati su entrambi gli schermi: così una stessa subroutine potrebbe funzionare in entrambi i casi. SCALE, quindi, va messo dopo un comando GRAPHIC. Può essere cambiato più volte tra un comando GRAPHIC e un altro.
Il programma SCALE offre un'illustrazione abbastanza chiara dell'effetto di questo comando, disegnando e ridisegnando alcune forme mentre si aumenta la scala. Una spiegazione è offerta nel programma MODI GRAFICI.

SCNCLR **Istruzione** **12, 128**

Sintassi:
SCNCLR [*modo*]

Abbreviazione:
sC

Svuota lo schermo. Le indicazioni della System Guide sono tutt'altro che chiare. SCNCLR da solo ripulisce lo schermo attuale. Se si sta usando lo schermo a 80 colonne, SCNCLR da solo ripulisce tutti gli schermi.
Se si esegue prima il comando GRAPHIC 0 lo schermo grafico non sarà ripulito.

SCNCLR 0 ripulisce lo schermo a 40 colonne
SCNCLR 1

SCNCLR 2
SCNCLR 3
SCNCLR 4 puliscono tutti lo schermo grafico e/o il testo a 40 colonne (2 e 4) e la RAM colori (3 e 4).
SCNCLR 5 pulisce lo schermo a 80 colonne.
SCNCLR 0 e SCNCLR 5, quindi, per i due schermi, equivalgono a PRINT"*S*" ("*S*" corrisponde al tasto CLR).

SCRATCH **Comando** **16, 128**

Sintassi:
SCRATCH "*nomefile*" [,D*n*] [,U*n*]

Abbreviazione:
scR

Cancella un file dal disco. Cioè lo elimina dal direttorio, senza eliminare la registrazione del file, ma svincolando quella parte del disco, che potrà così essere occupata da un nuovo file.
Dopo aver dato il comando in modo diretto, il calcolatore presenta la domanda:

ARE YOU SURE?

Rispondere Y per eseguire, battere un altro tasto per annullare.
Da programma, la cosa avviene "silenziosamente" e richiede quindi un po' di attenzione, ma può essere utilissima. Per esempio, immaginate un programma che raccoglie dei dati in un file "prov", poi li alfabetizza e li salva in un nuovo file "def". A questo punto, nell'interesse dell'ordine e della pulizia, SCRATCH "prov", dato nel programma, elimina il file che non serve più. Analogamente dopo CONCAT (B$) TO (A$) il programma può eliminare B$ con SCRATCH (B$). SCRATCH"*" ripulisce TUTTO IL DISCO
SCRATCH"File.*" elimina tutti i file che cominciano con "File."
SCRATCH"*=p" elimina tutti i programmi.
Si veda la voce DIRECTORY per altri esempi applicabili anche a SCRATCH. Se si vuole usare SCRATCH con i caratteri speciali ? oppure *, è un'ottima idea eseguire prima il comando DIRECTORY usando lo stesso argomento che si intende poi eseguire con SCRATCH: questo dà la possibilità di confermare che verranno eliminati soltanto i file da distruggere e non altri. Dopo uno SCRATCH errato, se ci si accorge prima di effettuare altre operazioni di scrittura, è possibile restaurare i file eliminati dal direttorio con l'opzione RESTORE FILES del dischetto CBM DOS SHELL. Per sostituire un file usare prima SCRATCH e poi DSAVE.

SGN — Funzione — 20, 16, 64, 128

Sintassi:
v=SGN(x)

Abbreviazione:
sG

Dà il segno della variabile x:

1	il segno è +
0	il valore è 0 (né + né −)
−1	il segno è −

SIN — Funzione — 20, 16, 64, 128

Sintassi:
v=SIN(x)

Abbreviazione:
sI

Restituisce la funzione trigonometrica seno dell'argomento x, che rappresenta un angolo in radianti.

SLEEP — Comando — 128

Sintassi:
SLEEP *n*

Abbreviazione:
sL

Ferma il programma per *n* secondi, dove *n* va da 1 a 65535. Il tasto STOP lo interrompe in caso di necessità. Utile per esempio per presentare una serie di schermi con scritte senza alcun intervento da parte dell'utente per passare allo schermo successivo, o per evitare che l'utente proceda troppo velocemente tra un INPUT e un altro.
SLEEP 5 ferma tutto per 5 secondi.
Per brevi ritardi, vale ancora il loop del tipo:

```
FOR T=1 TO 25:NEXT T
```

SLOW **Comando** **128**

Sintassi:
SLOW

Abbreviazione:
nessuna

Torna a una velocità di 1 MHz, riabilitando così lo schermo a 40 e la grafica. Si veda anche la voce FAST.

SOUND **Comando** **16, 128**

Sintassi:
SOUND *voce, frequenza, durata* [,*direzione*] [,*fr.minima*] [,*passo*] [,*forma d'onda*] [,*impulso*]

Abbreviazione:
sO

Suona con la voce specificata.

voce	da 1 a 3
frequenza	da 0 a 65535
durata	da 0 (nessun suono) a 32767 sessantesimi di secondo
direzione	fa salire (0), scendere (1) oppure oscillare (2) la frequenza rispetto a *fr.minima*; default 0
fr.minima	se si è specificato ,*direzione*, parte da questa frequenza minima
passo	cambia la frequenza in passi (da 0 a 65535); default 0
forma d'onda	si veda la voce ENVELOPE e la pag. 7-12 della System Guide.
impulso	si veda ENVELOPE (per *forma d'onda* 2).

Nella documentazione fornita dalla Commodore, manca una tabella che dia un'idea del valore in hertz (Hz) della frequenza. Il comando PLAY, comunque, elimina la necessità di determinare i valori esatti della frequenza per le note musicali. SOUND quindi va bene per musica "relativa", cioè non necessariamente legata a valori precisi in Hz, e per tutti gli effetti sonori.
Il programma SOUND sul dischetto che accompagna questo volume presenta una rassegna delle possibilità, che non può essere completa dato il numero enorme di combinazioni possibili. Poiché il programma, mentre

funziona, presenta sullo schermo i parametri, si consiglia di prendere nota quando si trova il genere di effetto che si vuole ottenere.

SPC — Funzione — 20, 16, 64, 128

Sintassi:
SPC(*n*)

Abbreviazione:
nessuna

Dopo PRINT o PRINT# inserisce *n* spazi prima di stampare il carattere successivo.

```
PRINT"LA"; SPC(10);"TERESA"
LA          TERESA
```

SPRCOLOR — Istruzione — 128

Sintassi:
SPRCOLOR [*c1*] [,*c2*]

Abbreviazione:
sprC

Definisce il valore di uno o entrambi i colori disponibili per tutti gli sprite in modo multicolor.
I valori di *c1* e di *c2* sono i soliti numeri per i colori (1-16). Si vedano anche RSPCOLOR e il programma SPRTUT.

SPRDEF — Comando — 128

Sintassi:
SPRDEF

Abbreviazione:
sprD

Questo potentissimo comando presenta uno speciale schermo-laboratorio sul quale si definisce uno sprite da ogni e qualsiasi punto di vista. È ampiamente descritto nel Capitolo 11 dedicato agli sprite.

SPRITE — Istruzione — 128

Sintassi:
SPRITE *n* [,*on*] [,*colore*] [,*priorità*]
[,*espx*] [,*espy*] [,*modo*]

Abbreviazione:
sP

Accende uno sprite, ne definisce il colore-base, la priorità, le espansioni *x* e *y* e il modo grafico (normale o multicolor).

n	numero dello sprite
,on	1 accende, 2 spegne lo sprite
,colore	valore da 0 a 16 che definisce il colore unico dello sprite quando non è in modo multicolor
,priorità	0 passa sopra gli altri oggetti sullo schermo: 1 passa dietro
,espx	espande lo sprite in senso orizzontale: 1 espande, 0 contrae
,espy	lo espande in senso verticale
,modo	multicolor=1, normale=0

Una trattazione più ampia è dedicata a SPRITE nei Capitoli 11 e 12. Inoltre la variazione di tutti i parametri è illustrata nel programma SPRTUT.

SPRSAV — Comando — 128

Sintassi:
SPRSAV *n, var$*
oppure SPRSAV *var$, n*
oppure SPRSAV *n1, n2*

Abbreviazione:
sprS

Immagazzina i dati per uno sprite in una variabile stringa:

```
SPRSAV 1, A$
```

La forma seguente, invece ha l'effetto opposto: copia i dati contenuti in una variabile all'area dedicata a un determinato sprite:

```
SPRSAV A$, 1
```

Oppure, per copiare i dati da uno sprite a un altro:

SPRSAV 1, 2

I byte che servono a definire uno sprite diventano una stringa binaria: è necessario evitare di usare PRINT con queste stringhe, perché i loro byte non contengono veri caratteri ASCII. PRINT# e INPUT# darebbero risultati imprevedibili. Perciò il sistema migliore per salvare queste stringhe è di salvarle in gruppi di 8, con BSAVE. L'alternativa è di usare MID$ e ASC per ottenere i singoli valori dei byte, salvare questi e leggerli con GET#.
Esiste un altro modo di produrre stringhe dello stesso genere (si veda la voce SSHAPE), nonché un altro modo di usarle (GSHAPE). Mentre BLOAD fornisce un ottimo metodo per conservare gli sprite su disco, con il grande vantaggio di creare un file indipendente dal programma, il tempo di caricamento del file sarà pur sempre tale da rallentare il programma stesso. SPRSAV dà la possibilità di caricare in memoria un *qualsiasi* numero di sprite prima ancora di avviare il programma. Per esempio in una matrice spr$(*n*) dimensionata per 240 elementi si potrebbero contenere 30 serie di 8 sprite, tutti disponibili in un lampo tramite una routine del tipo usato nel cartone animato alla fine del programma SPRTUT e discusso nel Capitolo 12 sugli sprite. Anche l'ultima parte del programma SHAPES (Menu Comandi Grafici del disco, Lato 1) illustra l'interscambio tra sprite e stringa.

SQR — Funzione — 20, 16, 64, 128

Sintassi:
v=SQR(*n*)

Abbreviazione:
sQ

Restituisce la radice quadrata di *n*, che deve essere >0.

SSHAPE e GSHAPE — Comandi — 16, 128

Sintassi:
SSHAPE *var$*, *x1*, *y1* [,*x2*, *y2*]
GSHAPE *var$*, [*x*,*y*] [,*modo*]

Abbreviazione:
sS, gS

Registra (SSHAPE) un pezzo di un disegno prodotto sullo schermo ad alta risoluzione in una stringa. La stringa, se definita in modo compatibile, può essere usata poi come sprite (tramite SPRSAV), oppure ricreata (ferma) sullo schermo ad alta risoluzione tramite GSHAPE.
GSHAPE può essere usato anche per visualizzare sullo schermo (in posizione fissa) uno sprite trasferito in una stringa mediante SPRSAV. Ne dà un'illustrazione il programma MICROCHAR.
Coordinate: sono influenzate da SCALE. Se non sono specificate *x2*, *y2*, l'angolo inferiore destro sarà il punto in cui si trova il pixel cursor.
Le *x*, *y* di GSHAPE specificano l'angolo superiore sinistro. Anche in questo caso il default è la posizione del pixel cursor.
Per conservare uno sprite SSHAPE deve registrare il numero corretto di byte, poiché uno sprite ne contiene solo 67, mentre SSHAPE può registrarne fino a 255. Per questa operazione è necessario che

$x2-x1=23$

Il seguente esempio dovrebbe essere sufficiente per chiarire il concetto:

```
SSHAPE SP$, 12, 12, 35, 32
```

usa le coordinate normali (senza SCALE). Disegnare un qualsiasi sgorbio entro le coordinate 12,12,35,32 ed eseguire questo comando. Adesso

```
GSHAPE SP$, 50,50
```

lo riproduce in un altro punto dello schermo. Mentre con SPRITE lo possiamo vedere in copia unica (se non creiamo 2 sprite identici), con GSHAPE lo possiamo mettere dappertutto.

```
SPRSAV SP$,1
```

crea uno sprite che possiamo poi muovere e, se vogliamo

```
SPRSAV 1,2
```

crea uno sprite 2 uguale al primo.
GSHAPE può essere usato anche per variare la forma salvata con SSHAPE. Ciò si ottiene tramite l'opzione [,*modo*]. Questo è un numero da 0 a 4:

- 0 riproduce la forma com'era
- 1 lo inverte (negativo)
- 2-4 eseguono le operazioni logiche OR, AND, XOR sulla stringa, produ-

cendo vari effetti. Per evitare una lunga spiegazione è opportuno provarle ripetendo

GSHAPE SP$,50,50, *n*

con $n=1, 2, 3, 4$.

Particolarmente utile è l'OR ($n=2$) che sovrappone il contenuto della stringa sul disegno esistente.
Se SP$ è stato salvato sullo schermo multicolor, queste operazioni possono produrre effetti strani. Il modo 0 è l'unico che dà risultati garantiti. Ma in tutti questi casi, nell'uso pratico si finisce per fare delle prove un po' empiriche. I programmi TAGLIACUCI e MULTICUCI (Shell Grafico) usano SSHAPE e GSHAPE per duplicare, spostare e/o cancellare una parte di qualsiasi grandezza di uno schermo grafico, mentre JOYDRAW e MULTIDRAW li usano analogamente per spostare zone grandi al massimo quanto uno sprite. Esiste inoltre il programma tutorial SHAPES (Menu Comandi Grafici, Lato 1).

ST **Variabile del sistema** **20, 16, 64, 128**

Sintassi:
x=ST

Abbreviazione:
nessuna

Dà lo stato corrente del sistema relativo ad operazioni di input/output (a eccezione dello schermo e della tastiera). Il suo uso è relativamente arcano, in quanto è necessario leggere la posizione di ciascun bit. I bit 0 e 1 riguardano il bus seriale; 2, 3, 4, 5 lettura/verifica da cassetta; 6 fine file o fine input dal bus seriale; 7 fine nastro oppure DEVICE NOT PRESENT sul bus seriale. Come esempio, una maschera per verificare se bit 7=1:

IF STATUS AND - 128 THEN PRINT "ACCENDERE IL DRIVE"

Ma è più comodo

IF ERROR=5 THEN PRINT "ACCENDERE IL DRIVE"

oppure, anche più semplice

PRINT ERR$(ER), oppur PRINT DS$

ST=64 (fine file) è il valore più utile con il C-128: lo usano per esempio i programmi TYPE e FIND per evitare di leggere oltre la fine di un file sequenziale la cui struttura non è nota.

STASH — Comando — 128

Sintassi:
STASH #*nbyte, intsa, expsa, expb*

Abbreviazione:
sT

È l'esatto contrario di FETCH: significa conservare, e prende *nbyte* dalla memoria principale del calcolatore e li trasferisce in un'espansione: dall'indirizzo interno *intsa* all'indirizzo *expsa* nel banco *expb*. Si veda il Capitolo 9.

STEP — Istruzione — 20, 16, 64, 128

Sintassi:
FOR...TO...STEP *n*

Abbreviazione:
stE

Stabilisce il "passo" del conteggio in un loop FOR...TO. Il numero *n* può essere qualsiasi valore, positivo o negativo, intero o frazione decimale. Si veda FOR...

STOP — Istruzione — 20, 16, 64, 128

Sintassi:
STOP

Abbreviazione:
stO

Blocca l'esecuzione del programma, presentando il messaggio BREAK IN LINE *N*. Ha lo stesso effetto di END (quest'ultimo, però, non presenta il messaggio). Utile nel collaudo di un programma, anche se con il C-128 possono essere usati invece TRAP, TRON e TROFF. Si veda anche la voce CONT.

STR$ — Funzione — 20, 16, 64, 128

Sintassi:
x$=STR$(x)

Abbreviazione:
stR

Converte il valore contenuto in una variabile numerica in una stringa nella quale i caratteri numerici sono preceduti da un segno – se il numero è negativo, e da uno spazio se è positivo. In entrambi i casi l'ultimo carattere della stringa è uno spazio. Così

```
PRINT STR$(15)
  15
PRINT STR$(-15)
 -15
PRINT STR$(1E5)
  1E+5
A$=STR$(33)
```

Usare RIGHT$(A$, LEN(A$)-1) per eliminare lo spazio iniziale.

SWAP — Comando — 128

Sintassi:
SWAP# *nbyte, intsa, expsa, expb*

Abbreviazione:
sW

Prende *nbyte* dalla memoria "interna" e *nbyte* dall'espansione specificata da *expsa, expb* e li scambia: ciò che era nell'espansione finisce nella memoria interna, e viceversa. Si vedano anche FETCH e STASH, e la discussione nel Capitolo 9.

SYS — Istruzione — 20, 16, 64, 128

Sintassi:
SYS indirizzo, [,*a*] [,*x*] [,*y*] [,*s*]

Abbreviazione:
nessuna

Accede a una routine in linguaggio macchina.
Solo in modo 128 le opzioni *a, x, y, s* specificano accumulatore e registri *x, y, s*.
Usare prima BANK per definire il banco di memoria

```
BANK 15: SYS 52684, 128, 5: BANK 0
```

TAB **Funzione** **2, 16, 64, 128**

Sintassi:
TAB(*n*)

Abbreviazione:
tA

Sposta il cursore di *n* posizioni verso destra a partire dalla prima colonna della riga attuale, a differenza di SPC che inserisce *n* spazi a partire dalla posizione attuale del cursore.
Premere il tasto TAB, in modo diretto, sposta il cursore 8 spazi verso detra: in "modo virgolette" produce un carattere in negativo che ha lo stesso effetto.

```
PRINT TAB(10) "TERESA"
          TERESA
PRINT "iTERESA"
          TERESA
```

La *i* indica il carattere speciale prodotto da TAB.

TAN **Funzione** **20, 16, 64, 128**

Sintassi:
v=TAN(*x*)

Abbreviazione:
nessuna

Dà la funzione trigonometrica tangente di *x*.

TEMPO **Istruzione** **128**

Sintassi:
TEMPO *n*

Abbreviazione:
tE

Discusso anche nel Capitolo 13. Determina appunto il TEMPO (velocità) di un pezzo musicale. Il default per *n* è 8. La durata di una nota intera (W) è data da

W=23,06/*n* secondi

Un numero >8 aumenta quindi la velocità, perché riduce la durata di W (e quindi anche delle altre note). I numeri più alti ($n^{max}=255$) sono in genere troppo alti per essere molto utili! Il programma TEMPO del Menu Comandi Musicali del dischetto illustra l'intera gamma di tempi.

TI **Variabile del sistema** **20, 16, 64, 128**

Sintassi:
x=TI

Abbreviazione:
nessuna

Contiene il numero dei sessantesimi di secondo trascorsi dopo l'accensione del sistema. L'utente non può variarne il valore, se non spegnendo e riaccendendo il sistema.

TI$ **Variabile del sistema** **20, 16, 64, 128**

Sintassi:
PRINT TI$

Abbreviazione:
nessuna

Orologio a 24 ore, controllabile dall'utente. Parte dalle ore 000000 all'accensione del sistema, ma può essere regolato con una stringa nella forma TI$="OOMMSS" (Ore Minuti Secondi). Per regolare l'orologio interno

sulle ore 09: 10: 00 impostare l'orologio, qualche secondo prima dell'ora voluta, con

TI$="091000"

Non premere RETURN fino a quando non siano esattamente le 9 e 10. In seguito,

PRINT TI$

produrrà l'ora esatta fino allo spegnimento del sistema. Naturalmente, dopo le 235959 torna a 000000. All'interno di un programma si può usare un INPUT per chiedere l'ora all'utente. Si potrebbe anche comprendere nel programma una subroutine (collocata in modo che il programma la esegua molto frequentemente) del tipo:

```
60000 if val(ti$)=>233000 then ?"g È ora di andare a letto":
goto 60000: else return
```

Questo interromperà il programma non appena TI$ supera "232959", e continuerà a suonare il "campanello" (*g*, CONTROL-G) e a stampare il messaggio fino a quando l'utente non prema STOP. Non usare un'espressione del tipo IF TI$="233000" perché questa funzionerebbe solo tra le 23:30:00 e le 23:30:01.
I programmi ELLISSE e ORA ESATTA (Menu Comandi Grafici del dischetto) illustrano l'uso di un INPUT per chiedere l'ora all'utente, e l'uso di CHAR per presentare l'ora sullo schermo (usando MID$ per dividere TI$ in 3 parti e stampare separatamente ore, minuti, secondi). Per misurare la velocità di un programma, mettere TI$ a 000000 con

```
10 TI$="000000"
(segue la parte del programma che si desidera controllare)
100 ?"Tempo richiesto:"; ti$
```

TO **Istruzione**

Si usa in varie espressioni, mai da sola (FOR..., GOTO, APPEND, COPY, RENAME, DRAW).

TRAP **Istruzione** **16, 128**

Sintassi:
TRAP *riga*

Abbreviazione:
tR

Discussa in un altro capitolo. Intercetta un errore (per evitare che si fermi il programma) e trasferisce l'esecuzione alla *riga*, che conterrà una qualche soluzione al problema, anche mediante la lettura delle variabili EL (linea dell'errore) e ER (tipo dell'errore) o della funzione ERR$ per visualizzare il messaggio relativo. Si veda anche RESUME.

TRON e TROFF **Istruzioni** **16, 128**

Sintassi:
TRON
oppure TROFF

Abbreviazione:
trO, troF

Dopo TRON e fino a un successivo TROFF, durante l'esecuzione di un programma verrà visualizzato tra parentesi quadre [ ] il numero della riga in cui si trova il comando che si sta eseguendo. Se quindi ci sono più comandi (o istruzioni, ecc.) nella riga, il numero di quella riga verrà stampato più volte.
Si può usare in modo diretto e poi eseguire il programma, ma se il programma non è proprio elementare, e se si ha almeno una vaga idea del punto in cui si cerca l'errore, conviene inserire TRON poco prima dell'inizio della zona sospetta, e TROFF poco dopo: così si manterrà lo schermo più leggibile. Infatti, lo schermo si riempirà di molti numeri che si aggiungeranno alle scritte volute dal programma. Rallenta il programma perché naturalmente la stampa dei numeri di riga occupa del tempo. L'argomento è trattato anche nel Capitolo 4 di questo libro.

USR **Funzione** **20, 16, 64, 128**

Sintassi:
v=USR(*x*)

Abbreviazione:
uS

Una specie di GOSUB che utilizza una routine in linguaggio macchina il cui indirizzo di partenza deve essere contenuto (per il C-128) nelle locazioni 4633 e 4634. Bisogna quindi collocare l'indirizzo di partenza della routine in queste locazioni prima di usare USR. Se necessario si userà BANK per cambiare banco di memoria. La routine effettua le operazioni richieste sulla variabile *x*, e il risultato (nell'esempio dato nella sintassi) diventa il valore di *v*.

VAL — Funzione — 20, 16, 64, 128

Sintassi:
v=VAL(*x*$)

Abbreviazione:
nessuna

Dà il valore numerico contenuto in una stringa, se questo viene prima di altri caratteri non numerici. Ignora lo spazio, CHR$(32). Se il primo carattere non è numerico, restituisce 0. Con un po' di fantasia, si possono usare INSTR, MID$, ecc., per estrarre un valore da altri punti all'interno della stringa.

```
10 a$="200 Fiat 500"
20 print val(a$), val(right$,a$,3)
run
200        500
```

VERIFY — Comando — 20, 16, 64, 128

Sintassi:
VERIFY "*prog*" [,*n*][,*ril*]

Abbreviazione:
vE

Controlla se un programma o altro file binario è stato registrato correttamente su nastro o su disco. *n* è 1 per il nastro, 8 per disco (ma per il disco si può usare anche DVERIFY). L'opzione ,*ril* è come per LOAD: indica

di verificare il programma a partire dalla posizione in memoria dalla quale è stato salvato.
Con il nastro è utile, anche se sembra strano, verificare un file inesistente: infatti, se si salvano i programmi con EOT (si veda SAVE), è un modo semplice per arrivare alla fine della parte già occupata del nastro, per poi salvare un nuovo programma o file di dati. Se si salva un programma e poi si cambia modo grafico, VERIFY terminerà con il messaggio VERIFY ERROR: infatti GRAPHIC richiede oltre 9 K di memoria all'inizio del BASIC. Ciò significa che il programma cambia posizione se si usa GRAPHIC o GRAPHIC CLR. Il programma può essere perfetto, ma il calcolatore non lo trova agli indirizzi nei quali si trovava quando fu salvato.
VERIFY "programma",1,1 - verifica "programma" su nastro a partire dall'indirizzo dal quale è stato salvato.
VERIFY "programma",8 - esattamente come DVERIFY "programma".
VERIFY "file",8,1 - verifica un file binario. Questo può essere un programma salvato con DSAVE o SAVE, oppure un file binario di dati salvato con BSAVE. Per quest'ultima operazione non è possibile usare DVERIFY.

VOL — Istruzione — 16, 128

Sintassi:
VOL *n*

Abbreviazione:
vO

VOL definisce con *n* (0-15) il volume assoluto delle tre voci sonore. Questo è diverso dall'opzione U dell'istruzione PLAY, che stabilisce il volume relativo (0-9) della voce indicata.

WAIT — Istruzione — 20, 16, 64, 128

Sintassi:
WAIT *indirizzo, n* [,*n*]

Abbreviazione:
wA

Attende fino a quando una determinata posizione di memoria non contenga *n*. Si usa per certe (arcane) operazioni di input/output: la più utile è l'attesa di un segnale in arrivo, per esempio dal modem, lungo un'inter-

faccia RS232. Il valore *n* è una "maschera". L'esempio dato nella System Guide

WAIT 36868, 144, 16

poiché 144=10010000 e 16=00010000, attende finché il bit più a sinistra (7) è 1 oppure il bit 4 è 0.

WAIT 211, 8

attende la pressione del tasto ALT.

WIDTH **Comando** 128

Sintassi:
WIDTH *n*

Abbreviazione:
wiD

WIDTH 2 raddoppia la larghezza di linee tracciate con DRAW, CIRCLE, BOX. 2 è il massimo: 1 (default) il minimo. Non ha effetto se la linea è orizzontale, a causa della diversa risoluzione. L'opzione W nel programma JOYDRAW utilizza WIDTH.

WINDOW **Comando** 128

Sintassi:
WINDOW *col,riga,col,riga* [,*clr*]

Abbreviazione:
wI

Definisce una finestra di testo (40 o 80), eventualmente usando l'opzione ,*clr*, per ripulire solo la zona definita (in pratica ,*clr*=,1: se non si specifica niente, non accade niente; è quindi inutile specificare 0 per questo parametro). I valori sono le due coppie *col,riga* (la prima indica l'angolo superiore sinistro, la seconda l'angolo inferiore destro). Spesso non si vogliono specificare entrambi gli angoli di una finestra (come in molti tutorial del disco): poiché WINDOW invece ne ha bisogno, si può usare PRINT CHR$(27)"T" per stabilire una finestra limitata solo in alto, e

PRINT CHR$(27)"B" per stabilirne una limitata solo in basso.
Il valore *col* va da 0 a 39 per lo schermo a 40 colonne e da 0 a 79 per quello a 80: *riga* va da 0 a 24 per entrambi. A differenza di CHAR, sullo schermo a 40 colonne NON è ammissibile un numero maggiore di 39.
Questo comando è trattato nel Capitolo 9, e il dischetto contiene due brevi tutorial, WINDOW e W2.

XOR — Funzione — 128

Sintassi:
v=XOR(*n1*,*n2*)

Abbreviazione:
xO

Dà il risultato dell'operazione logica OR-esclusivo su due argomenti *n1* e *n2*.
È noto anche come OR-esclusivo: La tabella della verità per OR è:

x	y	xORy
1	1	1
1	0	1
0	1	1
0	0	0

(Se uno o entrambi i bit=1, OR dà 1). Per XOR è invece:

x	y	XOR(x,y)
1	1	0
1	0	1
0	1	1
0	0	0

(Se uno ma non entrambi i bit=1, XOR dà 1). Si noti la sintassi completamente diversa rispetto a voci come AND e OR. Il tutorial ORANDXOR ne illustra le proprietà servendosi dello schermo grafico ad alta risoluzione: per esempio POKE *n*, XOR(PEEK(*n*),255) inserisce nel byte *n* il "contrario logico" del valore che conteneva (255=11111111). Sullo schermo grafico questo inverte l'immagine, producendone il negativo.

Contenuto e uso del disco

Vengono qui di seguito riprodotti i menu dei due lati del disco. Alcuni programmi sono spiegati anche nei vari capitoli del libro. Per utilizzare il Lato 2 è opportuno effettuare alcune copie del disco, nel modo spiegato al termine di questa appendice.

A.1 Menu Comandi Vari

A = MENU GENERALE
B = INSTR
C = PRINT USING
D = PUDEF
E = ASCII CBM
F = ASCII tabella
G = ALFASORT [ordinamento alfabetico]
H = BEGIN/BEND [2 programmi]
I = DO... LOOP
J = DIM
K = KEY
L = HEX/DEC
M = BIN/DEC/PIXEL [conversione binario/decimale]
N = OR/AND/XOR
O = BSAVE
P = RESTORE
Q = DIRTUT [versione in BASIC]

R = COMANDI BASIC DISCO [sintassi e appunti]
S = PEEK-POKE
T = TYPE [visualizzare il contenuto di un file]
U = FIND [cercare una stringa in un file]

A.2 Menu Comandi Musicali

A = MENU GENERALE
B = PLAY TUTOR
C = ACCORDI [2 programmi]
D = STRUMENTI
E = STRUMENTI 2
F = TEMPO
G = OTTAVE
H = BACH1
I = BACH2
J = SOUND TUTOR
K = ENVELOPE TUTOR

A.3 Menu Comandi Grafici

A = MENU GENERALE
B = CHAR
C = SPRITE TUTOR
D = GRAFICA TUTOR [sintassi]
E = COLOR TUTOR [uso delle fonti]
F = PAINT TUTOR
F = MODI GRAFICI [alta e bassa risoluzione]
H = DRAW
I = CIRCLE [2 prog]
J = BOX
K = SCALE
L = ORA ESATTA
M = ELLISSE [colori che "spandono"]
N = SSHAPE/GSHAPE [SHAPES]
O = MICROCHAR
P = WINDOW
Q = WINDOW 80 col.
R = 80 colonne
S = 80 colonne negativo

A.4 Menu Lato 2 (Grafica)

A = Terminare
B = DRAW
C = JOYDRAW •
D = MULTIDRAW
E = SALVASPRITE
F = SALVAGRAFICI
G = TAGLIACUCI modo GRAPHIC 1
H = TAGLIACUCI modo GRAPHIC 3
I = GRAFICI A BARRE/ISTOGRAMMI [alta risoluzione]
J = GRAFICI LINEARI [alta risoluzione]
K = SCATTERGRAM [alta risoluzione]
L = GRAFICI A TORTA [bassa risoluzione]
M = DISCOTUT [versione compilata di DIRTUT]
N = GALLERIA/LETTORE
.... INSTALL [non disponibile dal menu]

I programmi della serie DRAW e vari altri sono descritti nel testo. Su entrambi i lati del disco esistono altri file: in particolare file contenenti bit map di disegni (prefisso gr.), file contenenti informazioni sui colori dei disegni (prefisso cf.) e file di sprite (prefisso sp.). Il Lato 2 contiene anche file sequenziali di dati per i seguenti programmi:

- BARRE: diagrammi a barre ed istogrammi (prefisso gb.)
- LINEARI: grafici lineari (gl.)
- SCATTERGRAM: diagrammi di distribuzione (gs.)
- TORTE (prefisso gt.)

I programmi corrispondenti salvano i loro dati con questi prefissi e, naturalmente, hanno la capacità di presentare sullo schermo un direttorio selettivo contenente solo i file con il prefisso del programma. Hanno inoltre la possibilità di usare SALVAGRAFICI per creare file bit map (prefissi gr. e cf.)

A.5 Copie di lavoro del disco

Il dischetto che accompagna questo volume è un "flippy". Equivale, in altre parole, a due dischetti indipendenti che possono essere letti anche con i drive a singola faccia 1541 e 1571.
È comunque saggio preparare almeno una copia di un disco nuovo (se, come in questo caso non è protetto), ma in questo caso è addirittura ne-

cessario se si desidera sfruttarne tutte le possibilità.
In primo luogo, il drive 1571 sarà lento nella lettura del "flippy": richiede molto tempo per capire che non è un normale disco a doppia faccia. Mentre il 1541 non incontrerà naturalmente difficoltà di questo tipo, il Lato 2 del disco contiene molti programmi che devono effettuare operazioni di lettura e di scrittura: poiché lo spazio è interamente occupato dai programmi è necessario suddividere i programmi copiandoli su dischi diversi. Le operazioni da effettuare sono abbastanza semplici.

1. Usare il CBM DOS SHELL (dischetto fornito insieme al calcolatore) per copiare ciascuno dei due lati su un disco indipendente.
2. Controllare bene queste due copie (purtroppo il DOS SHELL non sempre produce copie perfette).
3. Preparare altre copie del Lato 2 e usare il programma INSTALL per eliminare alcuni programmi da ciascuna, in modo da disporre dello spazio necessario per creare file di lavoro.

Naturalmente le copie fatte con il 1571 sono a doppia faccia: ciò rende meno necessario preparare altre copie, ma sembrerebbe comunque opportuno disporre di uno spazio maggiore.
Per copiare il disco, accendere il calcolatore dopo aver inserito nel drive il disco CBM DOS SHELL. Seguire le istruzioni, selezionando COPIA DISCO sul menu.
Se si dispone del drive 1571 a doppia faccia, al termine dell'operazione si troverà che il direttorio sembra ancora quello di un disco a un solo lato, con un numero esiguo di blocchi liberi (BLOCKS FREE). Questo è dovuto a un errore nel DOS SHELL: fortunatamente è sufficiente eseguire COLLECT per disporre di un disco correttamente formattato, ed è indispensabile farlo prima di usare il disco. Il DOS SHELL non sempre copia il programma BOOT. Per controllare, inserire il disco nel drive e premere il tasto RESET sul lato destro del C-128. Se non si avvia il menu ("MENU.GEN" sul Lato 1, "GRA.MENU" sul Lato 2), il programma INSTALL (Lato 2, non disponibile dal menu) consente di ripristinare il BOOT, per entrambi i lati del disco.
Poiché può accadere che la copia sia imperfetta, si consiglia di provare i programmi del Lato 2, nonché i file grafici (usare GALLERIA: si veda più avanti). Se tutto è in ordine, usare questa copia per preparare le altre, conservando il dischetto originale in luogo sicuro.
INSTALL consente inoltre di preparare i seguenti dischi:

A. Grafica ad alta risoluzione (DRAW, JOYDRAW, SALVAGRAFICI, TAGLIACUCI, DISCOTUT)
B. Grafica a bassa risoluzione (MULTIDRAW, SALVAGRAFICI, TAGLIACUCI, DISCOTUT)

C. Grafica aziendale (LINEARI, BARRE, TORTE, SCATTERGRAM, SALVAGRAFICI, DISCOTUT)
D. GALLERIA (presentazione automatica di disegni, grafici, ecc.).

Preparare una o più copie del Lato 2 con il DOS SHELL, poi inserire la copia nel drive; inserire in modo diretto:

RUN"INSTALL

e seguire le istruzioni fornite. INSTALL elimina tutti i programmi non richiesti sul disco del tipo prescelto.
Sul menu di questi dischi specializzati appariranno tutti i programmi come sempre: se si tenta di eseguire un programma non presente sul disco, il menu presenta un messaggio di avvertimento.
Il Disco D (GALLERIA) sarà normalmente uno dei dischi A-B-C che si è riempito di grafici. INSTALL in questo caso eliminerà tutti i programmi, lasciando solo i file con prefisso "gr." o "cf.". Notare che i grafici prodotti dai programmi del Disco C devono essere stati salvati in forma di bit map (i file sequenziali di dati con prefissi gb., gt., gs., gl. non sono leggibili da LETTORE, ma solo dagli stessi programmi che li hanno prodotti).
Caricare GALLERIA dalla copia completa del disco tenuta come riserva, e usarlo per "convertire" il disco in "galleria d'arte". Per illustrarne il funzionamento, è stata creata una piccola galleria dimostrativa. Usando la copia completa del Lato 2, caricare GALLERIA dal menu e leggere le note esplicative; al termine di queste si trova un menu che consente di creare una "galleria" con catalogo, di aggiornare un catalogo esistente, oppure di prendere in visione la galleria esistente senza modificarla. Scegliendo quest'ultima opzione si presenteranno sullo schermo i cinque file grafici presenti sul disco, insieme ai titoli registrati nel file sequenziale CATALOGO.
GALLERIA è un "programma suicida": quando l'utente ha creato il proprio catalogo e ne è soddisfatto, elimina buona parte di se stesso, diventando un programma molto più breve capace soltanto di leggere il file CATALOGO.
Questo programma ha il nome provvisorio di LETTORE, e alla fine di tutte le operazioni sarà l'unico programma sul dischetto. Poiché il settore BOOT del disco si aspetta di trovare invece il programma GRA.MENU, si conclude la preparazione del disco con un innocente inganno:

RENAME "LETTORE" TO "GRA.MENU"

Un disco del genere può servire ad esempio per mostre, conferenze, vetrine, video, eccetera. Collegando l'uscita RF o video composito del C-128 a un videoregistratore si può creare una videocassetta.

Indice analitico

Questo breve indice non elenca sistematicamente tutti i comandi del BASIC 7.0 riportati nel Dizionario del BASIC, poiché questi sono più facilmente reperibili tramite l'ordine alfabetico del Dizionario stesso. In diversi casi la trattazione più ampia si trova nel Dizionario. I nomi dei comandi BASIC sono riportati in maiuscolo. I nomi dei programmi del disco sono invece seguiti da un asterisco.
Così, per esempio, APPEND *è un comando ma non un programma,* Dirtut* *è un programma ma non un comando, e* COLOR* *è un comando e il nome di un programma.*

W

X

Z

88 386 0041 4 W. Ettlin e G. Solberg, *Il GW-BASIC per Personal Computer Olivetti*

88 386 0042 2 D. Watt, *Il LOGO per il Commodore 64*

88 386 0043 0 T.J. Byers, *Guida al PC AT IBM*

88 386 0044 9 D. Kater e R. Kater, *Guida alle stampanti Epson*

88 386 0045 7 R.L. Tokheim, *Progetti di circuiti elettronici per microcalcolatori*

88 386 0047 3 C. Siechert e C. Wood, *Il manuale MS-DOS 3.2* (in preparazione)

88 386 0048 1 H. Peckham, *Programmazione strutturata in BASIC*

88 386 0049 X W.H. Murray III e C.H. Pappas, *L'Assembler per l'80286/80386* (in preparazione)

88 386 0050 3 D. Andersen, C. Cooper e B. Densey, *Il dBase III in pratica*

88 386 0051 1 D. Carroll, *Programmazione in Turbo Pascal*

88 386 0052 X P. Hoffman e T. Nicoloff, *Il manuale MS-DOS per Personal Computer Olivetti*

88 386 0055 4 J. Heilborn, *Guìda al Commodore 128*

88 386 0060 0 P. Robinson, *Programmazione in Turbo PROLOG* (in preparazione)

88 386 0063 5 A.E. Stanley, *Il BASIC 7.0 per il Commodore 128*

88 386 0601 3 S. Harrington, *Computer Graphics - Corso di programmazione*

88 386 0602 1 O. Lecarme e J.L. Nebut, *Pascal - Guida per programmatori*

88 386 0603 X M. McGilton e R. Morgan, *Il sistema operativo UNIX*

88 386 0604 8 E. Rich, *Intelligenza Artificiale*

88 386 0605 6 M. Schagrin, J. Rapaport e R. Dipert, *Logica e computer*

88 386 0606 4 W. Newman e R. Sproull, *Principi di Computer Graphics* (in preparazione)

88 386 0607 2 L. Hancock e M. Krieger, *Il linguaggio C*

La produzione software comprende:

88 7700 902 0 C.A. Street, *PROFILE 2 - Foglio elettronico integrato per lo ZX Spectrum*

88 7700 903 9 S. Nicholls, *Routines in Assembler per la grafica avanzata con lo ZX Spectrum*

88 7700 904 7 A. Bleasby, *Assembler/Disassembler per il Commodore 64*

88 7700 905 5 ACS Software, *ZX Spectrum Monitor*

88 7700 906 3 G. Fitzgibbon, *Projector 1 - Uno strumento per costruire presentazioni grafiche con lo ZX Spectrum*

88 386 0907 1 C. Opie, *QL Machine Code Editor/Assembler*

88 386 0908 X B. Thompson e W. Thompson, *Sistema Esperto McGraw-Hill per personal MS-DOS*

88 386 0909 8 A. Tal, *Generatore di lezioni per il Commodore 64*

88 386 0911 X B. Thompson e W. Thompson, *Sistema Esperto McGraw-Hill per Apple II*

La McGraw-Hill pubblica in tutto il mondo centinaia di libri di informatica per lo studio, la professione e il tempo libero. La produzione in lingua italiana comprende:

88 386 0001 5	J. Heilborn e R. Talbott, *Guida al Commodore 64*
88 7700 002 3	C.A. Street, *La gestione delle informazioni con lo ZX Spectrum*
88 7700 003 1	T. Woods, *L'Assembler per lo ZX Spectrum*
88 7700 004 X	R. Jeffries, G. Fisher e B. Sawyer, *Divertirsi giocando con il Commodore 64*
88 7700 005 8	G. Bishop, *Progetti hardware con lo ZX Spectrum*
88 7700 006 6	H. Mullish e D. Kruger, *Il BASIC Applesoft*
88 7700 007 4	N. Williams, *Progettazione di giochi d'avventura con lo ZX Spectrum*
88 386 0008 2	H. Peckham, *Il BASIC e il PC-IBM in pratica*
88 7700 009 0	H. Peckham, *Il BASIC e il Commodore 64 in pratica*
88 7700 010 4	S. Nicholls, *Tecniche avanzate in Assembler con lo ZX Spectrum*
88 7700 011 2	K. Skier, *L'Assembler per il Commodore 64 e il VIC-20*
88 7700 012 0	S. Kamins e M. Waite, *Programmazione umanizzata in Applesoft*
88 7700 013 9	A. Pennell, *Guida allo ZX Microdrive e all'Interface 1*
88 7700 015 5	P. Cohen, *Grafica e animazione con gli Apple II*
88 7700 016 3	C. Duff, *Guida al Macintosh*
88 7700 017 1	G. Kane, *Il manuale MC68000*
88 386 0018 X	P. Hoffman e T. Nicoloff, *Il manuale MS-DOS*
88 386 0019 8	E.M. Baras, *Come usare il Symphony*
88 7700 020 1	S. Nicholls, *Grafica avanzata con lo ZX Spectrum*
88 386 0021 X	L.J. Graham e T. Field, *Guida al PC-IBM*
88 7700 022 8	T. Field, *Come usare MacWrite e MacPaint*
88 7700 024 4	H. Peckham, *Il BASIC e gli Apple II in pratica*
88 386 0025 2	C. Morgan e M. Waite, *Il manuale 8086/8088*
88 7700 026 0	W. Ettlin, *Come usare il Multiplan*
88 7700 027 9	G. Mainis, *Il manuale ProDOS*
88 7700 028 7	J. Jones, *Il SuperBASIC del QL*
88 7700 029 5	C. Opie, *L'Assembler per il QL*
88 386 0030 9	W. Ettlin e G. Solberg, *Il BASIC Microsoft*
88 386 0031 7	D.L. Toppen, *Il Forth in pratica*
88 7700 032 5	R. Person, *Le meraviglie dell'animazione con gli Apple II*
88 386 0033 3	P.A. Sand, *Programmazione avanzata in Pascal*
88 7700 034 1	P. Hoffman, *Il manuale MSX*
88 7700 035 X	R. Person, *Le meraviglie dell'animazione con il PC-IBM*
88 7700 036 8	W. Ettlin, *Il Multiplan per il Macintosh*
88 386 0037 6	D. Kruglinski, *Introduzione al Framework*
88 386 0038 4	L. Barnes, *Come usare il dBase II*
88 386 0040 6	L. Barnes, *Come usare il dBase III*

Il C-128 costituisce la naturale evoluzione del famosissimo C-64 con il quale mantiene una completa compatibilità. La novità più significativa è rappresentata dal nuovo linguaggio, il BASIC 7.0, che comprende molte nuove istruzioni e comandi che lo rendono più potente e flessibile del tradizionale BASIC del C-64.
Il BASIC 7.0 per il Commodore 128 è ben più di un semplice manuale del linguaggio: oltre a una completa descrizione di tutti i comandi, funzioni e istruzioni, contiene infatti numerose routine e programmi di grafica e suono che dimostrano nella pratica le prestazioni della macchina e del linguaggio.
Tra gli argomenti più interessanti troviamo:

- la descrizione delle funzioni della tastiera
- la gestione delle periferiche
- le novità introdotte col BASIC 7.0
- il bit mapping e i comandi grafici
- le finestre
- gli sprite e l'animazione
- il suono

Conclude il libro un completo e documentato dizionario del BASIC 7.0 che costituisce, a tutt'oggi, il più completo testo di riferimento sul linguaggio.

Lire 26 000
(IVA 2% inclusa)

ISBN 88 386 0063 5